AGARD-CP-545

AGARD-CP-545

AGARD

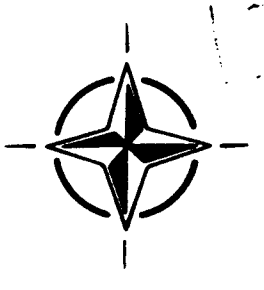
ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT
7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS 545

Aerospace Software Engineering for Advanced Systems Architectures

(L'Ingéniérie des Logiciels pour
les Architectures des Systèmes Aérospatiaux)

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.



NORTH ATLANTIC TREATY ORGANIZATION

Published November 1993

Distribution and Availability on Back Cover

AGARD-CP-545

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT
7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS 545

Aerospace Software Engineering for Advanced Systems Architectures

(*L'Ingéniérie des Logiciels pour* les Architectures des Systèmes Aérospatiaux)

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

North Atlantic Treaty Organization
Organisation du Traité de l'Atlantique Nord

The Mission of AGARD

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

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

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

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

Published November 1993

ISBN 92-835-0725-8

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

Theme

During the past decade, many avionics functions which have traditionally been accomplished with analogue hardware technology are now being accomplished by software residing in digital computers. Indeed, it is clear that in future avionics systems, most of the functionality of an avionics system will reside in software. In order to design, test and maintain this software, software development/support environments will be extensively used. The significance of this transition to software is manifested in the fact that 50 percent or more of the cost of acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered.

Thème

Au cours de la dernière décennie, bon nombre de fonctions avioniques qui étaient traditionnellement réalisées par des matériels faisant appel à des technologies analogiques le sont maintenant par des logiciels intégrés à des ordinateurs numériques. En effet, il est clair que dans les systèmes avioniques futurs la plupart des fonctionalités de ceux-ci résideront dans le logiciel.

Que ce soit pour la conception, les tests ou la maintenance de ce logiciel, un très large appel sera fait aux environnements de support/développement de logiciel. L'importance de cette transition vers le logiciel est attestée par le fait qu'au moins cinquante pour cent des coûts d'acquisition et d'entretien des systèmes d'armes avancés sont directement liés à des considérations logicielles. Il est aussi significatif que cet assujettissement apporte une flexibilité sans précédent, permettant d'adapter rapidement les systèmes avioniques à des situations de menace et des exigences opérationnelles en constante évolution.

Vu l'importance capitale des logiciels pour les systèmes d'armes, les pays membres de l'OTAN attribuent de plus en plus de ressources R&D à l'examen de tous les aspects de la science et de la pratique du logiciel.

L'objet du symposium était de réunir les experts en logiciel aérospatial militaire de tous les pays membres de l'OTAN, afin de permettre une mise en commun des résultats des activités de R&D dans ce domaine, et virtuellement tous les aspects de la conception des logiciels étaient abordés.

Avionics Panel

Chairman: Col. Francis Corbisier
StChef HK Comdo Trg & Sp LuM
Quartier Roi Albert Ier
Rue de la Fusée, 70
B-1130 Bruxelles
Belgium

TECHNICAL PROGRAMME COMMITTEE

Chairmen: Mr John J. Bart
Technical Director, Directorate of
Reliability and Compatibility
Rome Air Development Center (AFSC)
Griffiss, AFB, N.Y. 13441
United States

Dr Charles H. Krueger Jr
Division Chief
WL/AAA
Wright Patterson AFB, OH 45433
United States

PROGRAMME COMMITTEE MEMBERS

Mr T. Brennan	US
Ing. L. Crovella	IT
Dr R. Macpherson	CA
IPA O. Fourure	FR
Ir H. Timmers	NE
Mr R. Wirt	US

AVIONICS PANEL EXECUTIVE

LTC. R. Cariglia, IAF

Mail from Europe:
AGARD—OTAN
Attn: AVP Executive
7, rue Ancelle
92200 Neuilly-sur-Seine
France

Mail from US and Canada:
AGARD—NATO—AVP
Unit 21551
APO AE 09777

Tel: 33(1)47 38 57 65
Telex: 610176 (France)
Telefax: 33(1) 47 38 57 99

Contents

TECHNICAL EVALUATION REPORT

by

ICA Laurent Guibert
Délégation Générale pour l'Armement
Direction des Contructions Aéronautiques
Service Technique des Programmes Aéronautiques
26, Boulevard Victor - 00460 Armées
France

INTRODUCTION

The 65th Symposium of the AGARD Avionics Panel was held in Paris, France, on May 10 to May 13 1993. The subject of this symposium was "Aerospace Software Engineering for Advanced System Architectures". The programme Chairmen were Mr John J. Bart, of Rome Laboratories, and Dr Charles H. Krueger, of Wright Laboratories, both from the U.S.A.

THEME OF THE SYMPOSIUM

The theme of the symposium was quite broad : to bring together software experts from all NATO countries to share the results of their research and development in every aspects of software engineering.

This choice of theme for the symposium was argumented through the obviously crucial importance taken by software in advanced weapon systems, since most of the functionality of future systems resides in it, providing in addition an unprecendented level of flexibility. The common consciousness of that has led all countries to intensify research and developement in that field.

But in fact, another underlying reason for such a theme is what has been called (rightly) by some sprakers "the software crisis". In some older times, software may have been considered as an easy way to implement some functions in systems. The advantages of doing so were claimed to be doability and flexibility : there were no envisionned limitation to the capability of software to realize a function and to be modified as wished, considering that the size of the functions that were implemented through software was limited.

Since that time, people have discovered that software engineering is a technology and as such, has limitations inherent in the state of its art. But these limitations are not palpable, not precise. They are not, as for semi-conductor technologies, related to physical features of a given state of the art. They are not tied to a known and measurable parameter, such as the complexity of a specified algorithm or more broadly of a software individual module. In fact, that complexity is much less a driver than the size of the whole software system, in relation with the level of reliability (or safety, or quality ...) one wants it to reach.

The fact that the limits are not quantifiable may lead to forget that they exist : This can explain the problems that have been encountered for many airborne systems. This may also explain the figures given by one of the speakers, Mr Bergey from the NAWC (US), that perfectly demonstrated what is indeed "the software crisis" : an american study has come to the results that amongst a large number of software systems, only 5% were delivered and utilized without change or with minor changes.

In that situation, the theme of the symposium suggests that the "technology of software" includes all steps of work that participate to a software product, from the first specification from the user to the final validation / qualification testing. This has been confirmed by many speakers.

GENERAL DESCRIPTION

The symposium consisted of 6 sessions. The first four were dedicated to the typical steps of software engineering (according to a waterfall or a "V" model), i.e. specifications, design, realization, validation and testing. Session 5 was on software management and session 6 on software environments. 37 papers were to be presented, amongst which 3 have been cancelled (without any notice).

In his welcome address, Ingénieur Général Vrolyk, of STTE (FR), has clearly introduced the challenge of software engineering : software has to be considered relatively to its whole life cycle, including the

maintenance, in order to obtain a "total quality". This means rigorous methods for the entire life cycle, supported by a variety of adapted tools. Before trying to realize and validate a software product, one must validate the methods and processes.

Dr W. Royce, of TRW (US), presented then an outstanding keynote address. Dr Royce' explanation for the software crisis is that the methodologies are based on a requirement-first approach. Due to many reasons, the requirements are never frozen. The inherent (and theoretical) flexibility of software eases changes happening very often. Now there is a point, when this happens too often, where it defeats the theoretical ability of software engineering to tolerate the modifications. Once again, in that process, the technological limits are breaken. Dr Royce then proposes a brand new approach: architecture first, just-in-time-requirements. This would be the way for the future in order to really (re-) establish a high degree of flexibility, without having unacceptable consequences on cost, reliability, etc.

With that very accurate presentation, the symposium was really launched with a view to a future where the "crisis" has to be overcome, through an evolution of the software technology in order to push the limits away.

TECHNICAL EVALUATION

Session 1 - Software specifications

Paper 1, by Mr Chanet, was rather on methodology than on specifications. The main lesson was that when products evolve (here, the Airbus aircraft family), the increasing complexity and volume of software can be matched by adapting from one programme to the next one a sound methodology and powerfull tools. The speaker did not see a limitation to that process. It should also be noted that complexity of the software for that type of aircraft is lowered by the increasing complexity of the hardware architecture (number of black boxes and thereoff, of connections) : the size of each software unit remains reasonable. Another lesson was the necessity for the many cooperants to share a same "tool architecture", and not only a software architecture.

Paper 2, by Mr Paquot, described a method and tools for modelling, simulating and prototyping executable software specifications. The product (rotorcraft flight control system), method and tools are quite specific. The paper shows that such method and tools now exist and may work, although they have not been applied to a real project. Unreadable transparents made the presentation difficult to follow...

Paper 3, by Mr Borden, was an interesting presentation on the difficult problem of decision making processes. Here, the limitation are not related to the software itself, but rather to the hardware that may not be able to run the software. In such a case, where the limits are known and measurable, a strategy can be defined and implemented in order to obtain a product wich is not the best one possible, but which meets the essential part of the need (it is "sub-optimal") and is above all executable.

Paper 4, by Mr Cagnace, tackled the problem of airborne knowledge based systems. It showed that with new tools (particularly "XIA" , refered to as a real tecnological break-through), the expert systems can now be developped with the same kind of methodologies ("V" model) as other software systems. Expert modules can be developped separately and then "integrated" in the system together with classical software components (same philosophy as in keynote address, the framework being the inference engine). This was an encouraging paper, which let foresee as possible real knowledge based applications on board future aircraft, although some problems still need to be solved (particularly, the hardware performances).

In paper 5, Mr Beurrier described tools allowing to develop a potentially "zero fault" software for flight by wire system. The two solutions, a unique software versus severall versions in order to obtain the safety needed, have not been compared in terms of efficiency, costs, etc.

Session 2 - Software Design

A large proportion of the papers presented during this session dealt with the Object Oriented Design. This topic has in fact proven to be controversial.

Mr Occelli (paper 6) compared Object Oriented Design and Functional Oriented Design. His conclusion were that OOD was not THE solution to the software crisis, and that FOD was stronger for real time and safety critical products. A question was raised, wether OOD and Ada are compatible.

Mr Micoin (paper 7) described the advantages of a particular OOD method : HOOD. He stated, based on real programme experience, that the pair "HOOD-Ada" is workable and efficient, but has some lacks for real-time applications. He contradicted thus to some extent the former presentation.

Paper 8, by Mr Diemunsch, described the use of OOD for a very specific project. With this method, the C++ language has been used. C++ is recognized to be well adapted to OOD methods. Use of neuronal networks is to be noted.

The HOOD method was also the basis of work related in papers 9 (by Mr Mala) and 10 (by Mr Lacan). Paper 9 (quite complete) presented the advantages and drawbacks of HOOD : although it is recognized that it has weaknesses for real-time performances and analysis, Mr Mala would recommend using HOOD. Contrarily to paper 7, he stated the existence of a stuctural gap between the HOOD design and Ada code for some applications. Paper 10, however, did not raise this problem. In answering a question, the author said that

his experience was that HOOD could be efficient for real-time.

But there were also other areas treated. Paper 8a, by Mr Hameetman, described the experience of using a set of current off-the-shelf tools for a "small" product (9000 lines of "C" code) which needed a high degree of reliability (satellite). Advantages and drawbacks of the tools were discuused. This set of tools was deemed as adapted to a small team only.

Paper 11, by Mr Bosc, discussed the use of the LDS method, which is a CCITT standard, based on automats. The main advantage is the reduction of the testing effort. Here again, problems with real-time (communications between tasks) were evoked.

Paper 12, by Mr Hoebel, described a large scale programme of Rome Lab in the area of knowledge based engineering. Of most general interest is the effort on a KB software assistance toolset, that intend to assist in coordinating all software activities in a project and provide guidance to users in elaborating executable specifications.

The two first sessions have showed that a number of methods and supporting tools may be used succesfully (but one could question wether this represents the real world : the papers dicussed only successful projects...). A larger approach to software projects is needed, including the specification phase which is critical. No method seems to overcome the others in all areas. The future seems however to depend on very sophisticated tools, but will it be sufficient?

Session 3 - Programming practices and techniques

Paper 13, by Miss Noganno, presented a method used for prediciting an analysing coding errors in safety critical software. The method has not been implemented on large projects.

Paper 14, by Mrs Kean, described the Rome Lab's programme in software engineering technologies. Here appear some concepts that may be widely spread in the future, such as frameworks, re-usable software components, functional prototyping. Industry is closely involved in these activities, which is certainly a necessity in order to obtain usable tools, but is not sufficient for making these tools become recognized standards. A recognized standard necessitates that a large proportion of the industry uses it We must keep in mind that the defense industry is only a part of it, not necessarily large enough to ensure the success of a standardization process. This problem of standardization has been raised during the question session.

Paper 15 : cancelled

In paper 16, Mr Benjamin presented the Common Ada Run-Time System of the USAF. This interesting paper raises again the question of standardization, at another level. So many countries, services and companies develop their own Ada RTS : how can we ensure re-use and portability without a certain level of standardization? Why do'nt these questions be handled at adequate international levels? This does not seem to be a concern for the USAF, since CARTS is not intended to be proposed as a public standard.

Paper 17, by Mr Richard-Foy, describes a RTS addressing a subset of Ada for safety critical systems. This raises the same question as above...

In paper 18, Mr Gauthier introduced another newly developped operating system, that can be applied to an Ada RTS. It has been designed to solve caching problems encountered in real-time multiprocessor systems. This paper was technically interesting and accurate, but somewhat controversed in its hypotheses during the discussion. This shows again that one specific problem may have a number of solution.

This session witnessed the large number of developments and studies on going on coding, although coding is generally no more seen as the most acute problem in software engineering.

Session 4 - Software validation and testing

Testing is clearly a major aspect of software engineering. As stated by one speaker, it represents 25 to 50 % of the global effort. the major problem is (and has been for as long a time as software engineering exists) that a softw are system cannot be tested thoroughly, at least at the current state of the art. The questions are then how to do the right choices and how *far* to go?

Paper 19, by Mr Di Giandomenico, discussed the development of a data acquisition system for integration testing purposes and the problems encountered. The fast evolution of industrial standards and hardware were not the least ones. Is the solution in more "open" systems?

Paper 20, by Mrs Di Carlo, described a company-developped test toolset. The tools developped allowed a 20% reduction in testing effort. The basic finding was that the commercial off-the-shelf tools do not provide for thorough solutions for the software life-cycle in aeronautics. Will the aerospace industry be able to influence the market, or will they be continuously obliged to develop the tools and methods they need? The paper was rightly considered as "frightening" by the session chairman.

Paper 21, by Mr Stephenson, was a very comprehensive presentation of all the stages of testing a complex avionics system. State of the art processes and tools are described. Instructive. Testing the complete system, at late stages, is really a major challenge, and includes the fine measurements (propagation delays) necessary to finalize the weapon system.

In the continuity of paper 21, Mr Satterthwaite described then the different processes involved in testing airborne software programs. He established that thorough testing

is impossible and thus, testing requires comprehensive tools. Testing must be done against normal and abnormal situations.

Paper 23, by Mr Ward, presented a method for validating safety critical software. It was based on experience on a real project. The use of a formal specification language (Z), together with the classical static and dynamic testings, allowed to obtain a safe product (but not fault free). To be noted that the software was "small" (500 LOC). It was affirmed that using Ada tasking with its full features is quite impossible in such a system.

Paper 24 was cancelled.

Paper 25, by Mrs Gordon, was a hymn to motiving the teams in charge of testing and validating a software system (applied to the F22) : discipline and know how are necessary, but state of mind and organization are too.

The lessons learned during this session were numerous. Older techniques and tools are now outpassed and unefficient. Perspectives for testing is in improving the efficiency, probably not in reducing the costs. There is no response today to the problem of safety critical software (except for small ones). Testing must be envisaged in the earlier stages of a programme. And so on : everyone can find in that very rich session his proper lessons.

Session 5 - Software management

Paper 26 : cancelled.

Paper 27, by Mr Nairn, proposes a new Project Support Environment designed by the DRA (UK) to make the software systems independent from the developpers and from specific tools or environments, through its entire life-cycle. This environment, oriented towards the engineering description, is proposed for standardization, rather than the technology oriented standards such as PCTE or CAIS. It has been tested by transcription of an existing programme, but not yet used on a real project.

Paper 28, by Mr Brammer, proposes to extend the software engineering maturity model, developped by the Software Engineering Institute, to build a system engineering maturity model. Modifications should be minor. The framework for this model is not complete.

Paper 29, by Mr Sandrelli, dealt with quality improvement through classical hierarchical breakdown structure, configuration management and documentation. Project management is improved by timely and standardized information diffusion, through a data base.

Paper 29a, by Mr Bergey, presented a product oriented approach to the software project management (based on the US Navy SPORE model). The main problems the author sees in software engineering are quality and ability to be modified. But the primary driver is specification : if they are bad, the product will be bad. This is the GIGO process (Garbage In, Garbage Out). He stated that the quality payoff is cost saving. SPORE allows to obtain full visibility of the product, at each stage of the development. SPORE, in its last version, has not yet been used on a real project.

This session has demonstrated the awareness of the importance of management. Solutions really applied are "classical" and for the future, many concepts are studied. Type of management is in fact intrinsically tied to the history and culture of the system builder. Will the customers be able to change that and to impose their vision?

Session 6 - Software environments

Lack of adequate tools has been considered a major problem some years ago. This is no more the case today. But linking together the different tools supporting the different activities of software engineering is still one.

Paper 29b, by Mr Goodwin, illustrated this. After a presentation of the EuroFighter Software Development Environment, which is based on a framework, Mr Goodwin had a look to the future. According to him, based on the EF experience, many problems are unsolved. Software engineering is a part of a wider activity (system). Re-use, prototyping, automatic code generation are to be developped. Tools reamin a necessity and have to become stable, in order to avoid large modification in the life-cycle (commercial tools are highly volatile, due to the market). Multinational cooperation necessitates integrated, common tools and this is not recognized enough. Using different methods make errors possible at each interface and should not be envisaged, although not impossible. This was a somewhat pessimistic (but realistic) paper.

Paper 30, by Mr Fraboul, was more optimistic. It described quite clearly a tool allowing to programming fault tolerant applications distributed on parallel harware architecture. Laboratory results were excellent. This is crucial for the future, because avionics architectures will be more and more parallelized and distributed (see Pave Pace, ICNIA, ASAAC,...).

Paper 31, by Mr Cheratzu, decribed the AIMS Eurêka project. This is an industrial project aimed at improving methods in a "pragmatic" approach. One of the areas concerned is the collaborative work in multi-partners programme : this shows that progress are needed in that field... AIMS is in its demonstration phase (on real part of various international programmes). Up to now, results claimed are mainly a more common understanding of the developments between the 3 companies. Significant results are foreseen, after an integration phase.

Paper 31a, by Mr Corbin, described an environment dedicated to distributed systems with low bandwith communications, such as missions (multi-)simulators. Ada lacks in areas of dynamic binding and inheritance

can be overcome. So Ada weaknesses are not a problem?

In paper 32, Mr Coglianese presented a DARPA approach to future software engineering. It consists in separating an avionics system in different domains and allowing to specify the system in domain-specific languages, in order to obtain and exploit re-usability.

Paper 33, by Mr Pitete, described the Entreprise-2 open framework. Entreprise-2 features generic capacities for managing a software project, including a data base for management of re-usable components, and is able to integrate or to lodge specific tools for supporting the activities during the whole development. This is probably the kind of existing framework that should be advatageously standardized. This project reveals a new governmental approach in the area of software tools : to develop them and to encourage the contractor to commercialize them. Specific tools for the defense are expensive to maintain and upgrade, and defense budgets can no more afford that.

Session 6 was well balanced between actual environments and future ones. In some cases, future seems to be a ... long term one.

General comments

Generally speaking, the symposium has met the objective to give an overall view of the research and developments in software engineering within NATO and to share the results. It was well balanced between state of the art experience in elaborating software products and developments for future programmes. In general, the impression given was that papers on present projects showed a lot of questions to be solved and papers on future projects did not anticipate any, which can be considered as optimistic.

AVP does not provide the participants for evaluation forms, in order for them to give their view on the relevance of the papers. Personally, to a quite severe scaling, I rated 15 papers as excellent or very good (near 50%, which is very good as far as I am concerned), 10 as good, 7 as satisfactory and only 2 as poor or irrelevant.

The state of the art was very well described, with accurate highlights on the advances and the remaining problems. Globally, the papers presented a situation were the problems related to actual software projects are or can be solved, which is encouraging. But in an other hand, the situation is also worrying. Some papers contained figures of results obtained in terms of efficiency or productivity. These figures were between 20 an 50% after consequent developments. What in that case will be the efforts needed in order to cope with an order of magnitude of growth for future projects, with exponentially increasing complexity?

Developments for future programmes were necessarily tackled in terms of aims and objectives, since results are to come. These objectives were individually pertinent, but they revealed different directions of effort so it cannot be ensured that the sum of individual results, if the objectives are met, will enable resolution of all problems, because of bad compatibility between them. This is a traditional problem in that field : software developpers are led to make choices in terms of methods and tools, and every choice has its adavntages and drawbacks. This does not seem to change in the near future.

In addition, there were no paper (but one, n° 30) given by university (or university-like) researcher. Although it can be recognized that real experience is a basis for orienting research and developments, many "good" ideas were born in the universities. The links with these bodies should be kept alive!

The discussions allowed after the presentation of the papers were extremely pertinent and interesting. In fact, much of the very problems of software engineering were tackled during the discussions. For instance, re-use was a topic well and usefully adressed in questions, and also the need for methods and tools for formal proof of the software and for tool standardization.

If I had to find a cause for criticism, I would point out that specific problems related to future avionics architectures, which are much more integrated, although modular, than now, have not been considered enough. Indeed, these problems (distributed architectures, dynamic configuration and automatic reconfiguration, etc) will also be critical for the future airborne software systems. If this point has been taken into account by the Keynote Speaker, it was not the general case.

RECOMMENDATIONS

In this section, the aim is no more to evaluate the symposium, but to draw the lessons on the problematics of software engineering, as described during the symposium and with the assumption that the sum of the papers give an exact perspective of the state of the art.

The general impression given is that "the" solution to the software crisis is not for the near future. Individual solutions to specific problems allow advances of tens of percent (when and where measured) while the complexity and size of future airborne software systems are anticipated to increase by one or more order of magnitude.

It appeared often in the papers that the problems encountered were linked to the hardware architecture taken into account, and furthermore that new architectures (at system, sub-system or processor level) could make these problems obsolete. Probably is it symptomatic of one of the causes of the software crisis : software is thought in close relation with existing hardware architectures and it lacks the necessary hindsight in order to anticipate on the future. This may

in part explain that software technology is in a constant manner 5 to 10 years behind the hardware.

Software engineering is still today envisaged as a mean to resolve some issues (i.e. it is a way to realize some functions on platforms) much more than as a technology intrinsically necessary for the systems and which must thus be developped as a peculiar basis. Avionics systems are considered primarily in terms of hardware capacity to run software. Being given the pre-eminence, which is recognized, of the role of software in meeting the requirements and the fact that most of the problems during systems developments are now directly or indirectly related to (application) software, the necessity to think future systems upfront in terms of available software technology must be assessed. This underlined clearly the Keynote Address, but would be, as from many papers, a revolution in the way systems are envisionned today.

In order to summarize, there seem to be two keys for future systems : costs and specifications.

Costs have to be dra- -ly reduced, and to that end, complexity and size should be handled through methods and tools. Of particular importance are re-use and automatic coding. Re-use has, as a method, not yet really started to be exercized. According to some authors, re-use can only be envisaged at lower levels of software realization, but that point was controversial. But in fact, there is no real experience on large sacle re-use today. Automatic coding is also restricted to some narrow areas. It can be extended to automatic testing, although the real problem of software formal proof is far from being solvable. Particular emphasis should then be put on these topics.

Specifications are another heavy problem, because in one hand they are never as accurate as needed and in the other hand, when trying to obtain executable specifications, they become more and more difficult to elaborate. Methods and tools aimed at allowing users to elaborate accurate and executable specifications should be developped.

Another important point is related to the lack of standardization, both for methods and tools. This a severe repercussions on portability and re-use, but also on maintenance, since le life-cycles of tools are much shorter than those of airborne software products. Apart from Ada as language, there is no real standard within NATO. One reason is that there is no "perfect" product to be standardized. But, in one hand, there will never exist "perfect" methods or tools (without any lack) and, in the other one, one don not need that a product be perfect in order to have it become a standard. Ada is not a perfect language, since it has some lacks in real-time (easy to surpass today) and for safety critical software, but it has so many advantages in other areas that it is a usefull and recognized standard. At every level, given the axiom that perfect product will never exist, standardization is far better than nothing! It will allow, in particular, to concentrate the efforts aiming at improving methods and tools in a narrower field, which will make them less expensive and more efficient. It is thus recommended first to establish the adequate levels of standardization for methods and tools in software engineering and second to select or develop the related standard product, with the aim and the will to impose their use in all NATO real-time programmes.

KEYNOTE ADDRESS

REQUIREMENTS ENGINEERING BASED ON AUTOMATIC PROGRAMMING OF SOFTWARE ARCHITECTURES

Winston Royce
Systems Integration Group
TRW
One Federal Systems Park Drive
Fairfax, Virginia 22033-4411, U.S.

- o The Problem
- o An Emerging Solution
- o Some Exploratory Examples

Software development methodologies are generally based on a requirements-first approach.

Business considerations require at some early point in a software development that the requirements ought to be frozen.

But they never are.

Why aren't software requirements frozen?

- The acquisition group demands mission-oriented changes.
- The user group demands user-oriented changes.
- When other subsystem elements fail system performance is preserved primarily through software fixes.
- As software builders incrementally fix their initial design weaknesses they make requirements changes.

All of the foregoing are usually unforeseen in the beginning.

Software's flexibility inherently supports change particularly with respect to requirements.

Requirements changes - particularly those late occurring in the development life cycle - are troublesome because they defeat a business-like approach.

Software's easy flexibility (combined with its logical complexity) also amplifies its intolerance to faults.

Software's easy flexibility is simultaneously its premier strength. Software's flexibility is the best mechanism for adapting a system to its unforeseen, long-term future.

A major problem then is imposed on software developers.

How can we exploit software's flexibility for prolonging system life yet quickly build a low cost, fault tolerant initial system?

There is an emerging solution.

I would term it:

Architecture First - Requirements Second

What does "architecture-first" mean?

An executing architecture is built very early in the life cycle for all development participants to use.

Requirements come later.

The requirements oriented consequences of an early, executing software architecture are:

- Performance requirements are based on executing software not conjecture.
- User requirements are not attempted until an executing software infrastructure exists supporting user operations.
- (Most) requirements, can be safely changed at anytime, even late in the life cycle.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

How to control the increase in the complexity of civil aircraft on-board systems

P.Chanet V.Cassigneul
System Workshop
AEROSPATIALE
A/DET/SY M0101/8
316, route de Bayonne
31060 TOULOUSE Cedex 03
FRANCE

1. SUMMARY

After showing how the complexity of digital systems is doubling about every five years, and evoking the difficulties caused by this evolution, a methodological approach is described called the "Systems Development Workshop" aiming to gain mastery of this evolution.

Among design, production and validation stages, the importance of the design process is emphasized. The most acute problems of system development are generally described, and examples are given of the power and benefits that can be expected from computer design tools.

System architecture design and functional specification are expanded somewhat, to show what benefits can be expected from an integrated approach. Through the SAO example, all development stages are evoked.

A rough outline of the necessary capacities for a common work environment is drawn.

Finally, it is noted that the increasing necessity of international cooperation in civil aviation consolidates the proposed approach.

2. ONBOARD SYSTEMS : INCREASING COMPLEXITY

The complexity of digital systems on board civil aircraft is constantly increasing.

Figure 1 shows how, for AIRBUS programs, this complexity of on-board systems has doubled at each program : A310, A320 and A340. That is, every 5 years, while a constant reduction in the volume of electronics required to perform a given function has made it possible to keep the overall volume of the onboard electronics more or less the same, mostly thanks to VLSI (Very Large Scale Integration) technologies.

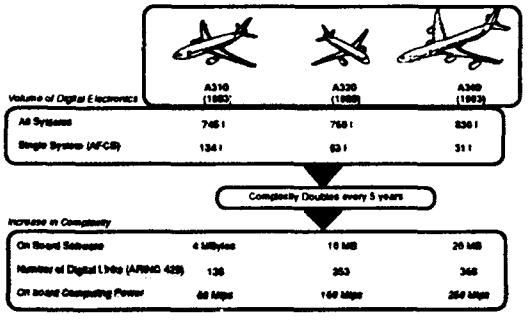


Figure 1 - Evolution of digital system complexity

Examining the reasons for the rapid growth in the complexity of the systems, one can be sure that this will continue for the next 10 years :

The role of on-board computers is steadily increasing ; their impact on system architecture is still moderate. Software developement techniques for safe, real-time systems are getting mature, together with multiplex data links. This allows for a certain amount of independance between functions and their supporting harware. The trend is towards de-localization of the functions within homogeneous hardware ("modular avionics"), making the overall functional architecture all the more complex and difficult to regulate.

Conversely, computing power and software versatility make new functions available for the aircraft. In the '80s, digital systems made "fly-by-wire" controls and electronic instrument panels possible, with their increased flexibility and safety features. In the '90s, they will help control the structural flexibility phenomena of wide body aircraft, with active control to improve passenger comfort. In addition, one must foresee the development of navigation and communications systems incorporating such new functions as GPS (Global Positioning System) or greatly enhanced ground/aircraft dialog through digital links : ATM (Air Traffic Management) and ADS (Automatic Dependent Surveillance). Mastering these additionnal functions is of course a great commercial asset for an aircraft manufacturer.

The ever increasing level of interconnection between systems will inflate the volume of information exchanged. This is a natural development mainly linked to the automation of secondary tasks, allowing the pilot to concentrate on flight control.

Last but not least, onboard systems are getting an ever greater influence on flight safety, thus requiring tight controls of design and manufacture (design-for-safety,...).

3. CAN THE DEVELOPMENT OF COMPLEX SYSTEMS BE MASTERED ?

The more complex a system is, the more difficult it is to keep within cost and schedule. Can the development of very complex, safety-constrained systems really be mastered ? The Airbus and ATR experiences of last are proof that it is, even in the context of international cooperation.

What are the main problems plaguing the development of complex systems ? Most common problems are :

- cost and time to get each function work !
- technical coordination, and associated difficulties in integrating the system

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

- how to make it dependable and to certify it after it is developped ?

Let us develop somewhat on these problems.

3.1. the infamous "Make-and-debug" approach

A lot of experience is the field of on-board systems, and not only in aeronautics, has shown how costly a "design, make and debug" approach can be. Debugging actions imply numerous and late design changes. The later the change, the higher the cost and schedule penalty. Furthermore, no validation action is possible in this approach before a first set of equipment has been produced, at full cost.

A well-known example, is the Flight Management System (FMS), the debugging of which required several thousand design changes, ending up with a certification delay of several years with respect to the original schedule.

Systems with intensive interaction with man are the most difficult to tune, regardless of their intrinsic complexity. Only a constant dialog with the future users (crews, etc...) *from the pre-project phase on* will allow for a satisfactory definition of these systems.

Our aim is therefore clear, we must :

- minimize the number of design changes needed,
- discover them and embody them at the earliest possible stage,

How can we get out of the habit of making-and-debugging ?

Quality insurance offers a set of solutions : design critical reviews, communication and updating procedures for overall consistency,...

The mastery of accurate models of physical phenomenons (aerodynamics, structural stress, electricity and electromagnetics, fluid dynamics...), of complex real-time simulation, and of automatic code generation open another set of solutions : early prototyping and/or simulation, if possible with *no physical equipments*.

The chosen approach becomes :

whenever possible, substitute early-stage design review or simulation to the more costly design-make-and-debug approach.

We will give examples of the benefits of such an approach later on.

3.2. Industrial cooperation & technical coordination

Airbus or ATR programs are carried out in international cooperation. Design offices must coordinate on a common design, with the help of a very light coordination structure.

The sheer volume of the design makes such a coordination quite difficult.

For instance, the length of the electrical wiring installed on an aircraft totals 180 km ; production of the electrical drawing set and its management is a formidable task indeed when 3 or more design offices contribute to it !

The logical definition of interfaces between systems is even more complex, especially when several versions are being developped simutaneously.

Ensuring the consistency between all teams is a matter of defining a common design reference, of configuration control and of communicating early and accurately to the right people all changes made or requested.

- *Interfaces between systems :*

This is one of the most difficult problems to master, first because of the very large number of data and information exchanges which exist and also because of the asynchronous operation of the various computers.

To address the interface problem, Aerospatiale has made up a data bank of signals exchanged in the aircraft in order to obtain centralized management. The resulting functions are :

- description of the signals exchanged,
- management of interfaces between equipment,
- consistency check on these exchanges.

The second problem concerns the "dynamic" interface of the electric signals exchanged. The asynchronous operation of onboard computers can generate temporary disagreements, which are often troublesome for the crews, especially in the case of electrical transients and on automatic reinitialization of one of the computers. The large number of equipment manufacturers, each with their own interpretation of the exchange rules (ARINC 429 standard) only increases the difficulty. Today, this problem is dealt with by a consistency check on all the signals exchanged between computers.

By addressing this problem from the design stage, we know how to reduce the number of spurious indications and/or unwanted disconnections of the onboard systems.

- *Definition and installation of electrical wiring :*

; the same applies to the routing of these cables which must comply with stringent segregation and installation rules.

To deal with this task, we have developed a tool called CIRCE (Conception Informatisée et Rationalisée des Câblages Electriques : Rationalized and computerized design of electric cables) which allows :

- wiring diagrams to be produced in CAD,
- the list of cables to be manufactured to be extracted in computer file form,
- these cables to be allocated to the assemblies (harnesses) and to their subassemblies (which correspond to manufacturing operations),
- data bank management of :
  - standard items,
  - diagrams,
  - cables,
  - cable terminations,
- the routing of the electric cables to be defined in 3D CAD.

The latter function, associated with automatic calculation of bundle lengths allows the requirements of the

Production, Inspection and Product Support departments to be met from the design stage.

3.3. Dependability & Certification

Dependability has (at least) four compoents : safety, reliability, availability, maintainability.

Achieving dependability.

Unmastered, the search for dependability is a close variant of the "make-and-debug" approach, as certification and/or dependability analysis, *at the end of a program*, make then a sudden, unexpected demand for numerous and deep design changes.

Dependability must be achieved all along a project. It is a continuous process and an integral part of the design, manufacture, validation and in-service operation stages.

Without attempting to summarize in a few lines all the tasks allowing this dependability to be obtained, we can mention the most significant aspects.

This is a discipline which reflects the will to control the technical, human or environmental hazards. It is based, not only on the experience acquired (regulations, standards), but also on the ability to prevent risks by predicting them.

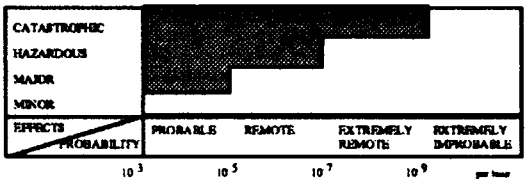


Figure 2 - Domain of compliance of events : consequences versus probability

- *During Design*

The prediction of functional failures and the assessment of their consequences allows :

- the criticalities of the functions to be defined,
- safety objectives to be allocated to the functional failures expected, thus to functions themselves, and then to their underlying equipments.

Dependability is then used as a basis for the architectural design of the systems, and for quality assurance, by allocation of performance objectives to the items in a system : this procedure was used for the first time on Concorde.

The prediction of external aggressions (lightning, icing) or specific hazards (engine burst, bird strike) makes it possible to define :

- the installation directives, (segregation,...)
- design precautions, (shielding of cables exposed to lightning...)
- circuit protections (guards protecting the circuits against running fluids...)

Validation and Demonstration

Demonstration of dependability is often a matter of putting together various analysis (i.e. failure mode analysis) and test results, in a somewhat "bottom-up" approach, for comparison to dependability objectives allocated in a top-down fashion.

Quality assurance takes all its importance here, by warranting that the necessary actions are taken to keep track of the objectives and to get the necessary measures. It is often enforced through regulation, for example, for onboard software, stringent quality assurance measures, based on the RTCA DO 178 document, are applied.

Safety analysis provides a good example of the demonstration problem.

Two tools are available and widely used today within the scope of the Airbus, ATR and Hermes programs :

The first tool named SARA (Safety And Reliability Analysis) is dedicated to collection and synthesis of safety data at the system-level.

The second tool named DAISY (Dependability Alded SYnthesis) allows the SARA-analyses of each system to be linked, ensures their coherence and deduces the global safety level of the aircraft.

In-service support :

Dependability participates in maintaining the continued airworthiness of the aircraft : the airworthiness follow-up allows, with hindsight, the hypotheses made during the design phase to be confirmed. This leads to a typical problem of corporate memory, as follow-up data keep trickling back during the 20 to 25 years of an aircraft-type lifespan.

4. SYSTEMS DEVELOPMENT WORKSHOP

Mastering the development of complex systems appears to draw on 4 main capacities :

- the ability to simulate or review specifications early,
- the ability to communicate and to ensure consistency all along the development between all partners, while preserving beneficial independance (concurrent, asynchronous engineering).
- the ability to assign performance objectives (for instance dependability objectives) to all items and to track and document these objectives through multiple versions of the product definition,
- the ability to produce updated documentation in pace with the technical work.

This can only be achieved with a rational and methodological approach to system-development within a organized structure. In Aerospatiale, this structure is called the Systems Development Workshop.

4.1. Definition

"The systems workshop is the coherent set of methods and tools required for optimum development of systems within a given aircraft program context".

As we do not start from scratches, the System Workshop approach must be an integrative approach, bringing together existing tools and methods into a common methodological framework.

It must observe the practical industrial organization, and provide means of communication and reference data-bases

in accordance to the actual coordination networks and rendez-vous.

A formal expression of specification will be highly regarded, since it give all 4 major capacities : ability to simulate, unambiguous communication, configuration and performance management, and easy documentation.

On the other hand, the recommended approach is pragmatic rather than systematic : care must be taken not to impair the natural flexibility of the industrial organization through too much formalism. For instance, the overall mission definition of civilian transport aircraft is stable enough from a program to the next that it would not be advisable to enforce a complete and strict functional analysis at the aircraft level.

4.2. Domain covered :

The domain can be roughly estimated by examining the stages required to deliver a "ready to go" system.

A conventional "V" presentation of development activities is given on Figure 3. There are three main stages: design, manufacturing and validation.

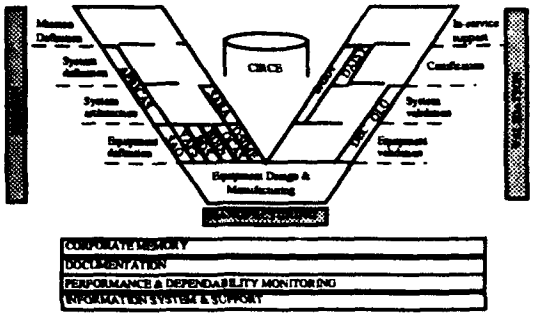


Figure 3 - System development cycle and associated tasks

In the V presentation, a validation step is associated to each design step. Thus presented, the domain of the system workshop is vast as it covers the three above mentioned phases.

It must be kept in mind that theses stages are neither truly chronological, nor independant. Most of the times, the various activites are carried out in parallel or in close ineration. Additionally, the model should actually picture nested "V"s.

For instance, Equipement Manufactiring appears as a single activity from the aircraft manufactured point of view, but it is actually itself sub-structured into design, manufacturing and validation stages.These intermediate phases are not part of the *primary* domain of the System Workshop, as we tend to sub-contract them entirely ; they are part of the System Workshop visibility domain however, as one must be capable of following the progress of the manufacturing activities and ensuring technical traceability to the System level.

The primary domain of a System Workshop should accordingly be adjusted according to industrial organization and usual worksharing and subcontracting boundaries.

In Aerospatiale, it covers System Definition, System Architecture and "Equipment Definition" (including detailed specification of major functions), System Validation, and Certification (more accurately, the system part of the aircraft certification).

4.3. Focusing on the Design phase

Requirements on the system workshop can be collected by reviewing the three main stages above.

The best way to describe a Systems workshop would be to review the various stages of the process (*DESIGN, MANUFACTURING, VALIDATION*) to examine, for each of these phases, the corresponding requirements and the solutions that can be found.

Special emphasis is placed here on the DESIGN stage as this stage is critical for the final product quality level ; within this stage, it will not be possible to examine all the steps and therefore all the tasks comprising each of these steps : only the most significant ones will be considered.

System definition

First of all, the functions that a system must fulfilled must be defined, and basic technology choices made. This results in the primary "system definition", with a rough functional breakdown.

System architecture design

In a first stage, the criticality levels of each of the functions must be specified.

From this point on, the system architecture design can be undertaken considering :

- aircraft specific constraints
- available (and chosen) technologies,
- the natural grouping of functions,
- the cost, weight, overall dimension and maintenance objectives, etc...

Re-use of the system architecture of previous aircraft as a starting point for the new one is a common and cost-effective practice in civil aeronautics. This method cannot apply, though, when developing a brand new system, or when using emerging technologies. As an example, Integrated Modular Avionics (I.M.A.) can call for a complete re-design of system architecture. In this case, the possibility of using common resources for several functions requires that the distribution of the hardware resources in the aircraft, and the allocation of the functions in this hardware be tackled together.

A tool dedicated to system architecture design is being experimented at Aerospatiale. Its name : AFRICAS (Analyse Fonctionnelle et Répartition Interactive pour la Conception d'Architectures Systèmes : functional analysis and interactive allocation for system architecture design) is representative of the aim pursued. The tool allows to describe concurrently the intended functional and hardware breakdowns of the system. It then supports for each function a redundancy choice based on its criticality level, and the allocation of each redundant "instance" of the function to a suitable equipment, keeping track of available resources and enforcing design rules. The resulting "architecture" can be assessed against dependability requirements, or simulated (qualitative simulation) to assess the impact of specific configurations or failure conditions. Then, if necessary, the architecture can be amended.

Equipment specification :

This phase could more aptly be split into "Specification of System Functions" and "Equipments definition".

An item of equipment can be defined from two types of specifications, either a general specification which lists the functions and performance required, or a detailed specification which describes the execution logics of these same functions.

The former is what is usually called "Equipment definition".

The latter can (and will increasingly) be carried out at the system level rather than at the equipment level, with the introduction of IMA and other distributed-system technologies.

Aerospatiale has chosen to specify in detail the items of equipment intimately linked with the operation of the aircraft, that is mainly, the critical equipment : fly-by-wire computers, autopilot, man/machine interface, etc...

Originally, the specifications were written in natural language ; the overrichness of this type of language led to software coding errors, as the specifications could be incorrectly interpreted by the programmer analyst.

With this in mind, Aerospatiale developed a complete specification and functional validation workshop based on a tool named SAO (Spécification Assistée par Ordinateur, Computer Aided Specification).

The major originality of this tool is its graphical language which uses a symbol library and assembly rules known by all electronics and automation engineers. This language covers the field of operational logics and closed-loop systems ; an example of a sheet with SAO formalism is shown on Figure 4.

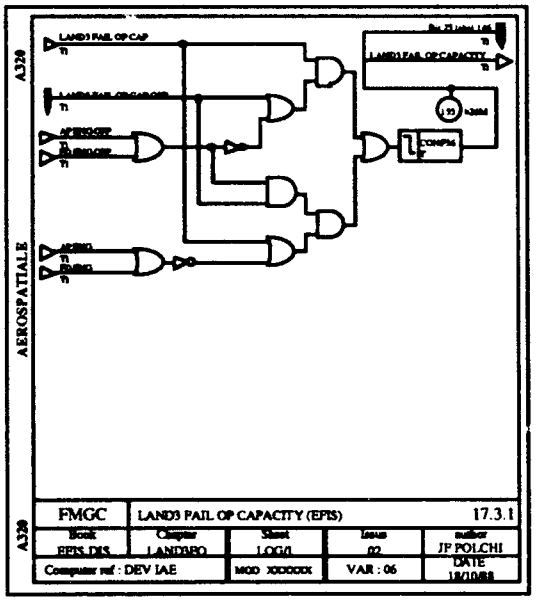


Figure 4 - SAO : a computer aided specification "sheet"

This formalism allows :

- the specified function to be readily understood without ambiguities, which avoids most coding errors,
- changes to be controlled and therefore the traceability of the specification to be ensured,
- a consistency check to be conducted on each sheet, then on the specification.

Last, SAO allows automatic coding. This is not the least of its merits, as automatic coding has the following advantages :

- reduction in coding errors to a level which is practically null,
- elimination of unit testing,
- reduction in the software manufacturing cycle, especially the modification cycle,
- possibility of validating the functions of an item of equipment as early as the design stage. This point is covered by the following paragraph.

The VAPS tool (Virtual Anything Prototyping Systems) made by the Canadian company VPI (Virtual Prototype Incorporation), associated with the SAO tool allows cockpit display symbologies to be defined with prototyping and animation of symbols.

This task results in a specification for the equipment concerned. It is made up of :

- a part specified under SAO and/or VAPS :
  - flight control laws and/or operational logics,
  - equipment input/output signals,
  - display symbologies.
- a more conventional, mostly textual part concerning, among other things :
  - safety requirements,
  - physical characteristics,
  - environmental constraints,
  - etc...

Validation of specifications :

First of all, for reasons of clarity, we shall distinguish between validation and verification.

The verification of a product consists in ensuring its compliance with its specification. The validation operation consists in ensuring that the *product specifications* are correct and complete. The importance of this operation can be easily seen as it would be pointless to know how to code a specification automatically, without making errors, if the specification itself contained errors or was incomplete.

As we have already seen, this validation operation is covered by the right-hand, bottom-up part of the V. It is usually carrried out with such expensive means as ground-based integration rigs, aircraft simulators and flight tests.

Although integration rigs or flight test are invaluable in validating the completed systems and smoothing out all problems of interaction with the "real" world, they are quite oversized and impractical in the early tuning-up of

the logics of a system or in validating the consistency of its functions.

SAO provides a much cheaper and easy-to-use way to validate functional logics and consistency. Drawing up on its capacity for automatic code generation, simple, "desktop" simulation tools can be built, allowing the designer of a function to "fly" and validate "hands on" the resultant control software - within minutes ! If one of the design parameters is not satisfactorily met, re-iterating is only a matter of hours - as compared of a matter of weeks or months if a new test equipment had to be produced.

OCAS (Outil de Conception Assistée par Simulation : Simulation-assisted design tool) is a mini-simulator of this kind. It can code and run fly-by-wire and autopilot control laws in connection to the updated model of aircraft structural and aerodynamic response. A control panel, including mini-flight controls and simulated Primary Flight Display and Navigation Display can give a realistic "look and feel" of the future aircraft controls, while all control laws parameters can be monitored.

This tool therefore allows the aircraft control laws to be validated at an early stage in the development cycle, saving long hours of simulator and flight tests and many design changes.

OCAS does not take into account the hardware aspects - for instance failure monitoring and redundancy management.

These aspects are taken care of by another tool, OSIME, which can simultaneously simulate the operation of five computers, each computer including dual redundancy (or a "Command/Monitor" architecture).

- failure-free operation at tolerance limits,
- operation with failures and downgraded modes,
- cross-computer synchronizations,
- transient effects,
- feedback loops with simulated servocontrols.

This tool, which is extremely powerful, allows several hundred cases to be dealt with overnight.

Here again, automatic coding of the SAO sheets allows the modification cycle to be reduced to a few hours.

This tool is known by the name of OSIME (Outil de SImulation Multi-Equipements : Multi-equipment simulation tool). A representation is given on Figure 5.

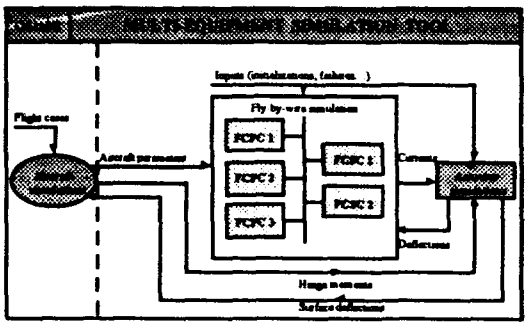


Figure 5 - OSIME

Manufacturing stage

In Aerospatiale's industrial organization, as Equipment Manufacturing is mostly sub-contracted, the System Workshop only has a visibility and work follow-up objective for this stage.

We make a notable exception of some safety-critical or highly aircraft-dependent software packages such as the fly-by-wire control laws.

Here, we make full advantage of SAO capacity for automatic code generation, with a Software Workshop connected to the System Workshop. Several such Software Workshops exist now in Aerospatiale, Eurocopter and some equipment/subsystem manufacturers, based on SAO and/or VAPS.

In the case of fly-by-wire software, an automatic code generator allows 70% of the software of the on-board computer to be derived from its SAO specifications. This tool had to be developed with the same degree of quality as embedded software.

AEROSPATIALE has also designed a test sequence-assisted generation tool. This tool is used to check that all the system functions specified in SAO language have effectively been coded (verification).

Last, a software configuration management tool identifies changes in the specifications and automatically controls the operations required to update the software accordingly.

Thus, by simplifying the manual tasks in the complete production cycle and by close coupling between the management of the "systems" specifications and the management of the software packages, AEROSPATIALE can comply with the level of quality required for its safety-critical systems.

4.4. Focusing on the Validation phase

We have already touched on this subject, as Verification and Validation (V&V) are ongoing concerns from the early design phases on.

- *Ground tests*

Distinction is made between :

- tests on partial benches which allow the operation of each of the systems taken separately to be tested with simulation of the peripheral systems,
- tests on the integration bench which allow the cross-operation of the systems to be tested. We try to make this integration bench representative of the aircraft as regards the geometrical shape, the cables, and the power resources such as the electrical and hydraulic power distribution and generation systems.
- tests on the flight simulator conducted to validate the aircraft control laws and the failure procedures in a real environment. This simulator is therefore equipped with a cockpit similar to the one on the aircraft with a simulated view of the outside world. Aerospatiale's experience on this subject shows that it is not necessary to move the cockpit to simulate the aircraft movements (for civilian transport aircrafts at least). This simulator therefore differs in this respect from the training simulators used in the pilot training centers.

These tests on simulators with pilots are the natural extension of those conducted on the "in-house" simulator called OCAS. In both simulation cases, the identity of the assessed software packages is ensured thanks to the use of

SAO and automatic code generation. On partial- and integration-benches, trimmed-down SAO specification are even used to simulate peripheral equipements.

- *Flight tests*

This is the ultimate phase. In an aircraft development cycle, this phase lasts approximately one year whereas the previous phases are, in general, spread over several years. The tests on the systems represent only a fraction of the 1200 flight hours required for the complete certification of the aircraft.

For this, the aircraft must be instrumented with high-performance recording systems in order to be sure that all transient phenomena are monitored. Several thousand parameters are thus recorded during each flight, either analogically for large bandwidth signals or digitally for the others ; the recorded data flow represents 200,000 bits per second of flight. This information is recorded onboard the aircraft but part of it is transmitted directly by telemetry to the ground allowing the test data to be processed in real time.

As a typical A340 flight can last over 10 hours, it is a amount equivalent to 7 GBytes of data that must be processed within 12 hours to extract anomalies and relevant data !

Once again, SAO can be of help. DECOLO (DECOnnexions LOgiques : logical disconnections) traces the real causes of abnormal results in logic functions. Searching for the causes of an unwanted warning or of an abnormal change of state of a system can be a real brain-teaser as, due to the asynchronous operation of the on-board computers, we cannot easily recover the time-sequence of the logics computation : distinguishing causes from effects can be a tedious cross-checking work. DECOLO does it in almost real-time using a simple backward analysis of the original SAO specification.

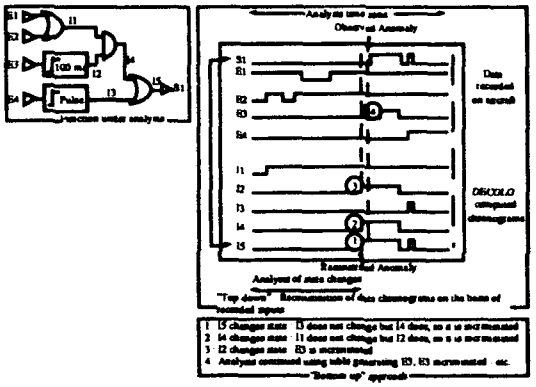


Figure 6- DECOLO: principle of table analysis for logics *disconnections*

4.5. Global tasks covering the whole development

Some tasks are active during the whole process.

Figure 3 identifies four of these tasks :

- *Corporate memory :*

Past experience or the company's memory must exist in an easily-accessible and user-friendly form, this is rarely achieved.

From the information required by the designer, we must single out :

- The technical directives and the regulatory texts : the pile of documents covering all the systems of an aircraft stands more than 6 feet high ! In fact, the designer needs a guide to find the information that he requires in all this undergrowth.
- The selective experience which results from actual debugging cases or even incidents or accidents. To perpetuate this experience, typical cases are volunteered and described "on the run" in a database, with entry forms kept as simple as possible. From that database, a team of experts extracts applicable 'Rules", with a procedure for following up the applications.
- The product knowledge : technical, factual data of course, but more important the "whys" and the "therefores", the technical justifications for the design,
- The know-how or, again, the expertise which makes up the company's culture and ways of reasoning ; for that reason, the System Workshop also includes a extensive program of ... mutual teaching or training.

Aerospatiale is developing a program called MERE (Mise En Règle de l'Expérience : experience set into rules) which is a global corporate memory project taking into account the various above mentioned aspects.

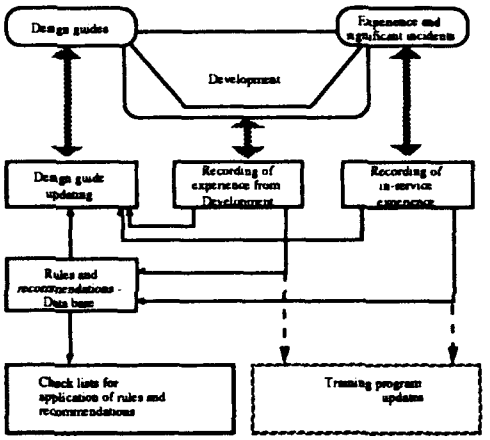


Figure 7 - Corporate memory organization : MERE : experience set into rules

Figure 7 explains the various components of this project. This perpetuation of the company's know-how seems to us to be fundamental ; it applies upstream of and throughout the complete design process.

- *Documentation chain :*

Today, most of the documents produced are not always derived from one another, there is no systematic chaining

between documents, nor between design and documentation.

Lacking methodology and standards, the documentation of complex systems soon becomes difficult to control.

To produce, manage and use the numerous documents which result from the design, manufacture and validation phases of a system, we need :

- a documentation model. This model can be built by analyzing the development process in term of activities, information produced by the activities and their use, and logical & chronological grouping of these informations ;
- document writing guidelines, and a common exchange format for structure documents,
- document chaining possibility,
- document configuration and information traceability management,

This program is basically in agreement with the aims of the CALS (Computer Aided Logistic Systems) initiative promoted by the US DoD (Department of Defence) ; most standards promoted by CALS, such as SGML (Standard Generalized Mark-up Language), are under study. SGML is already used in the Aerospatiale Product Support department, and is a basis for the future on-board Electronic Library System. Its use in support of design documentation is promising, but dependent on a thorough methodological analysis.

- *Dedicated system workshop host environment*

There is little difference between the host environement of a software workshop and that of a systems workshop. In both cases, the aim is :

- to supply a user-friendly interface in order to guide the user when accessing the various tools and functionalities of the workshop.
- to manage the objects which are :
  - proper to the workshop itself,
  - or produced by the workshop tools.
- to create or manage the links between all these objects :
  - either for the traceability of the activities,
  - or to make up the documentation.
- to activate the tools and support their communications
- to provide project management facilities with access rights,
- to communicate with other workshops (in particular, partners' and Equipment Manufacturers' Workshops),
- to support the company's procedures and terminology.

More than a piece of software, a System Host Environment is mostly an set of corporate standards. We cannot overemphasize the importance of harmonization and consistency as without it the designers' tools would flourish with the same logic as that governing the distribution of poppies along the roadsides in the springtime.

To develop this type of structure, we need widely-used standards and interfaces to be able to accommodate a large range of tools available on the market. For example, the PCTE (Portable Common Tool Environment) standard already used for software workshops.

This standardization is even more necessary as we are compelled to work in ever closer cooperation. This is the case for Aerospatiale, within the scope of the European civil aircraft programs such as Airbus and ATR.

5. ENCOURAGING RESULTS

A systems workshop is being gradually set up at the Aerospatiale Aircraft Division. At the time of writing, still a lot remains to be done, in particular at the systems host environment level. Also, the scope of SAO must be extended to new types of functions such as data bank management, scientific calculations, etc... and, on the other, by interfacing SAO with functional analysis tools capable of complementing it upstream.

In spite of these needed improvements, the contribution made by some of these tools can be mesured.

SAO

Figure 8 shows the improvements achieved.

Aircraft	A310	A320	A340
Number of Digital Units	77	102	115
Size of on-board software (MBytes)	4	10	20
Encoding errors for 100 kBytes	a few hundreds	a few dozen	<10

Note: On the A340, the systems benefitting from automatic encoding are :

- fly-by-wire : 70%
- Automatic Flight Control: 70%
- Display Computers : 50%
- Warning and Maintenance systems: 40%

Figure 8 - SAO impact of software coding errors

On the A310, when the specifications were written in natural language, there were hundreds of coding errors in software packages with a size of 100 Kbytes ; on the A320, thanks to the use of SAO, this number was reduced to a few tens of errors and, today, on the A340, by adding automatic code generation, the number of errors is less than ten.

The impact of OCAS and OSIME on specification errors is comparable, although no figure is available yet as these tools are currently being used on the A340/330 program.

These results show that the generation of complex software packages, which was rightly considered difficult during the last ten years, is today a problem well under control at Aerospatiale.

DECOLO

Automatic diagnosis of disconnections and/or warnings is an important step forward in the analysis of tests on

Systems with complex operational logics. This tool provides good results in 80 % of the cases, in less than 3 minutes of analysis compared with the many hours of manual analysis required on previous aircraft.

CIRCE

The contribution made by CIRCE affects two fields which are data quality and the development cycle :

- the quality of the definition of electric cables has been considerably increased to reach the following percentage : 99 % of the cable definition is correct following the first data processing operation.
- the processing cycle for a complete definition, has been reduced from 24 to 7 days.

Note that a CIRCE operation today calculates the definition of 230,000 different electric cables, each one of them being described by 50 parameters.

6 . CONCLUSION : REQUIREMENTS FOR THE YEAR 2000

Our aim in this document was to make the reader aware of the benefits of having a systems workshop to design, produce and validate complex systems.

Much more than an exhaustive description of such a workshop, the aim of this document was to promote the procedure leading towards this. Such an approach lives up to the aim pursued. Over and above the economical and lead time aspects, which alone justify the setting up of such a procedure, its fundamental strategic aspect must be underlined :

On the one hand, the communication and information transfer aspect, which is one of the goals of this project, is the key to multi-partner cooperations.

On the other hand, such a systems workshop is the cornerstone in the development of complex systems. With this asset, Aerospatiale is well on its way to mastering the complex systems of the year 2000 and beyond.

Discussion

Question S. GROISIL

Les logiciels de simulation et le logiciel de génération de code doivent-ils être certifiés? Si oui, comment le sont-ils?

Reply

La simulation est aujourd'hui un moyen d'efficacité interne et ne se substitue pas aux moyens de validation et de certification classiques.

Les chaînes de livraison et de génération de code automatiques, elles, ont un impact important sur la certification du logiciel résultant. La procédure acceptée aujourd'hui est de les "qualifier" sur la base des mêmes contraintes que le logiciel produit : application de la DO 178 à leur développement.

Question K. BRAMMER

You showed considerable decrease of coding errors for a given size of software, for the software of the Airbus family. Did you practice software re-use from one Airbus type to the next? If yes, can you comment on the contribution of re-use to the decrease in coding errors?

Reply

Re-use on object code, or even of programming language source-code has little meaning, as typical hardware components may evolve greatly from a programme to the next (5 years is typically the span of 1 or 2 generations of processors).

On the other hand, re-use of SAO specification is frequent, either through re-use and evolution (as from A320 to A340 or ATR) or through actual commonality (A340 and A330 specifications are mere versions of a single configuration environment, with extensive community of SAO "sheets").

Note that the library of SAO symbols is gradually growing. Each symbol actually stands for an agreed-upon software element specification and a re-usable software design. Defining a new library symbol is a very profitable -if heavy- investment :

1) as a standard element, agreed by both specifier and software engineer, it enhances the autonomy and efficiency of both,
2) its design and coding only have to be done and validated once in a programme - or for severall programmes - even though it may appear in hundreds of places in the specification. The level of confidence achieved in the overall library is such that unitary testing is no more required for most software versions produced,
3) discrepancies between simulation software and on-board software are minimal. They are mainly sequencing non-determinations and coding performance. The latter can be measured and corrected if necessary. The former can be specified (but beware of over-specification!).

Roughly, automatic code generation reduced coding errors to those present in the symbols - and a lot of work can be invested in their verification, so this is almost nil. The use of a library of symbols, stable from a programme to the next, makes software specification unambiguous and prevents most code-design errors.

Spécifications Exécutables des Logiciels des Systèmes Complexes

F. DELHAYE, D. PAQUOT
SFIM Industries Groupe Avionique
13, Avenue Ramolfo Garnier
91344 MASSY CEDEX
France

1. Résumé

Les technologies numériques ont pris une importance croissante à chaque génération de système de contrôle de vol. Le logiciel temps réel embarqué accomplit maintenant l'essentiel des fonctions de ces systèmes.

La nécessité d'industrialiser la production de ce logiciel s'est rapidement imposée. SFIM Industries a donc mis en place une démarche visant à outiller progressivement chacune des phases du cycle de vie.

Le pilotage des hélicoptères est caractérisé par une complexité intrinsèque. De plus l'architecture redondante des systèmes de pilotage augmente cette complexité.
La spécification de tels systèmes est donc une tâche très difficile. Or cette phase de développement revêt une importance particulière car les erreurs de spécification ne sont généralement détectées que dans les phases ultérieures du développement, ce qui induit des coûts de correction élevés.
Il convient donc de s'assurer au plus tôt de l'exactitude et de la validité de la spécification.

Dans le cadre de ses activités de Recherche et Développement en matière d'avionique, SFIM Industries a entrepris d'évaluer l'apport des techniques de spécification exécutable dans le cycle de développement des systèmes complexes.

Différentes techniques ont été analysées (analyse structurée, langages synchrones, systèmes experts) et parmi celles-ci deux ont fait l'objet d'expérimentation.

Ces techniques ont permis de construire des spécifications et de les valider avec des degrés de complétude plus ou moins importants.

L'intérêt de ces techniques a été clairement démontré. Elles apparaissent comme des éléments majeurs d'une démarche organisée de réduction du coût de développement des logiciels des systèmes complexes.

2. Glossaire Technique

.mode supérieur : asservissement de pilotage destiné à assurer le déplacement de l'hélicoptère sur une trajectoire dans l'espace.

.directeur de vol : fonction de visualisation permettant de donner des consignes de pilotage à l'équipage par l'intermédiaire de barres de tendance.

.axe de pilotage : axes de référence (tangage, roulis, lacet, puissance) autour desquels s'effectue le pilotage de la machine.

.trim : actionneur électromécanique permettant, à partir d'ordres électriques, la commande de vol de la machine.

.armement : l'armement d'un mode est une phase de préparation du mode durant laquelle le mode n'a aucune action sur le pilotage. Un mode armé ne s'engage effectivement que lorsque les conditions de capture sont vérifiées.

.capture : événement produit par la détection de la présence d'un faisceau de guidage.

3. Généralités

SFIM Industries, spécialiste mondial du contrôle de vol des hélicoptères, est une des premières firmes françaises d'équipements aéronautiques et de défense.
Elle développe dans ce cadre, des systèmes dont la complexité et la sécurité vont croissant.

3.1 Les problèmes de Qualité des logiciels embarqués

Aujourd'hui, presque tous les équipements et systèmes électroniques de bord utilisent ou utiliseront des calculateurs numériques.
Contrairement à leurs prédécesseurs de la première génération, ils sont maintenant intégrés dans des architectures entièrement

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

nouvelles et communiquent entre eux via des liaisons numériques.

Les possibilités offertes par les systèmes numériques permettent d'intégrer des quantités importantes de fonctions. De plus pour les systèmes de vol, la sécurité est assurée par la mise en place de systèmes redondants réalisant des surveillances croisées.
Tout cela conduit à une augmentation de la sophistication des systèmes.
La maîtrise de leur comportement est rendue difficile par l'accroissement des possibilités d'événements simultanés.

3.2 Le cycle de vie

Les coûts de développement d'un système peuvent être répartis sur les trois phases de développement suivantes:

A) A partir de l'expression de besoin du client, établissement de la spécification du système et développement de celui-ci conformément à la spécification (cycle de développement).

B) Correction de la spécification et reprise du cycle de développement jusqu'à obtension d'un système répondant aux besoins du client (recette ou mise au point opérationnelle).

C) Correction des problèmes trouvés en exploitation (maintenance).

- Le coût de la phase A est important, mais aujourd'hui bien identifié et maîtrisé lorsque l'on parcourt une seule fois le cycle de développement et de validation du système.

- Le coût de la phase C est mal cerné a priori, mais on espère toujours que la qualité du développement sera telle qu'il restera minime.

- Le coût de la phase B peut être très important et surtout difficilement maîtrisable.
Il dépend de dialogues entre différents intervenants et la formalisation est souvent difficile.
La sophistication des systèmes fait que la spécification ne reflète pas toujours complètement et suffisamment les besoins du client et ceci n'est vu qu'a la réception du système ou lors de la mise à disposition de l'utilisateur final.
Tout ceci conduit à reparcourir complètement le cycle de développement.
La remise en cause du travail effectué a de plus un impact psychologique non négligeable sur les équipes de développement.

C'est donc sur cette phase que nous avons décidé de concentrer nos efforts et pour cela mettre en place des moyens pour améliorer l'établissement de la spécification.

3.3 Définition des critères de qualité des spécifications

Une spécification est faite pour permettre la communication entre les divers intervenants du projet:
- -le client (celui qui finance).
- -le spécifieur (celui qui formalise le besoin)
- -le réalisateur (celui qui développe).
- -l'utilisateur (celui qui doit s'en servir).
- -le mainteneur (celui qui hérite des problèmes).

Chacun a une approche particulière de ce document :

le *client* et l'*utilisateur* ont un point de vue externe du produit, ils désirent voir réaliser le produit qu'ils ont imaginé.

le *spécifieur* doit par dialogue avec le client comprendre son besoin et le formaliser pour les équipes de développement.

le *réalisateur* et le *mainteneur* ont un point de vue interne du produit, ils désirent un document clair pour que les choix lors de la conception du système soient compatibles avec la destination du produit.

Il importe donc que cette spécification soit :

-structurée et homogène : une spécification est une référence et comme pour tout document de ce type elle doit être organisée suivant un plan logique qui facilite:

la lecture.
la consultation ponctuelle.
la modification.

-adaptée à ses lecteurs : les différents lecteurs ont des préoccupations et des cultures techniques souvent différentes. Il faut pourtant que la spécification permette à chacun de comprendre les objectifs, les contraintes, les possibilités existantes et les potentialités du système.

-cohérente : il est difficile de réaliser un produit dont la spécification est contradictoire.

-complète et sans ambiguïtés : il est nécessaire de ne pas laisser le concepteur faire des

interprétations ou des suppositions sur la spécification.

-conforme aux souhaits de l'utilisateur final : la spécification étant la première étape de la réalisation du produit, il est important que le spécifieur ait correctement traduit les désirs de son client.

Si la nécessité d'établir des spécifications avant de réaliser un produit est largement reconnue, les procédés pour conduire une telle activité restent peu élaborés et surtout très divers selon les environnements. La façon la plus répandue demeure la rédaction en langage naturel.

Face à cette approche intuitive il existe des méthodes, et des outils associés, venant assister le spécifieur.

Ces méthodes sont :

-les spécifications semi-formelles qui utilisent un "langage de spécification" textuel ou graphique, langage doté d'une syntaxe en général précise et d'une sémantique souvent assez faible, mais suffisante pour permettre l'automatisation de certaines vérifications.

-les spécifications formelles exprimées dans un langage à syntaxe et à sémantique précise, construit sur une base théorique solide et qui permettent des validations automatisées.

Elles permettent d'assurer la cohérence et la complétude de la spécification. Ce formalisme peut être suffisamment précis pour être interprété par l'ordinateur et ainsi fournir une spécification exécutable (maquette).

Cette maquette liée à une simulation de l'environnement du système peut permettre à l'utilisateur final d'observer le comportement de son futur produit.
Celui-ci peut donc, dès la phase de spécification, demander les rectifications qui lui semblent nécessaires.

L'établissement d'une spécification avec les méthodes envisagées sera réalisé au moyen de quatre activités

.Modélisation: activité de construction du modèle de besoins

.Simulation: activité d'exécution et de mise au point du modèle (aspects internes du modèle)

.Prototypage: activité de génération d'un code exécutable, image du modèle

.Validation: activité de vérification du modèle. Elle peut être faite soit sur le modèle, soit sur le prototype.
Pour chacune de ces activités un ensemble de critères a été identifié pour déterminer l'efficacité de la méthode choisie.

3.3.1. Activité de modélisation

la facilité de description : adéquation de l'outil à décrire des points particuliers de spécification, qu'ils soient séquentiels ou combinatoires.

la facilité de compréhension : aptitude du document généré à décrire une spécification pour un lecteur final non formé à la méthode.

la vérification statique de la complétude : possibilités offertes par l'outil pour détecter les erreurs et les manques.

la facilité de modification : le changement d'un élément de spécification ne doit pas entraîner pour le spécifieur une charge de travail disproportionnée.

la facilité d'évolution : l'addition de nouvelles fonctions doit être simple.

la qualité de la documentation produite : possibilité d'avoir un plan-type conforme à des normes, comme par exemple la DoD 2167A.

Cet ensemble de critères devrait permettre d'assurer la cohérence de la spécification et d'éviter les erreurs de la part du concepteur.

3.3.2. Activité de simulation et prototypage

Les critères associés à ces deux activités sont identiques.

la visualisation dynamique : interface graphique destiné à présenter ergonomiquement les données représentatives du système lors de la phase de vérification dynamique.

l'interfaçage avec des éléments extérieurs au modèle : possibilités offertes par l'outil de coupler plusieurs modèles entre eux, pour avoir la description complète d'un système composé de plusieurs sous-ensembles déjà modélisés.

l'interfaçage avec des langages de haut-niveau: possibilités offertes par l'outil d'interfacer le modèle avec des simulations d'environnement existantes écrites dans des langages de haut niveau.

le temps de réponse du système : rapidité de l'outil à calculer l'état système suivant.

Cet ensemble de critères devrait permettre de présenter au client une image fidèle de son futur produit et de pouvoir définir un protocole de recette du produit.

3.3.3. Activité de validation

la vérification dynamique du modèle : en mode interactif ou en mode "batch", simuler des actions sur les entrées du modèle et observer son comportement ainsi que son état interne.

la complétude des tests : vérification que chaque élément de spécification a été utilisé par au moins un test appliqué au modèle.

3.4 Rappel de la stratégie

Une étude a été menée pour rechercher des solutions industrielles à notre problème de spécification.
Pour simplifier le problème nous avons considéré que les méthodes de spécification couramment utilisées dans nos systèmes appartiennent à deux catégories.

- Description des aspects comportementaux (états du système)

- Description des aspects transformationnels (calculs et asservissements)

Cette dernière a déjà fait l'objet de travaux qui nous ont amené à développer, il y a quelques années, un formalisme graphique de description de ces aspects. C'est donc à la première catégorie que nous allons nous intéresser.

3.5 Présentation des méthodes évaluées

SFIM Industries, après avoir mené des études sur les différentes méthodes de spécification suivantes:

- Systèmes Experts (G2, KEE)
- Langages synchrones (ESTEREL, LUSTRE)
- Analyse structurée (ASA, TEAMWORK/SIM, STATEMATE)

a décidé d'évaluer plus précisément deux méthodes de spécification plutôt novatrices.

- La méthode de HAREL supportée par l'outil STATEMATE, développé par I-logix et distribué en France par Anticyp.

- Une méthode à base de système expert et de représentation orientée objet, supportée par l'outil G2 de Gensym, distribué par FRAMENTEC-COGNITEC.

3.6 Présentation de la cible support de l'évaluation

La cible applicative de ces méthodes a été fixée à un sous-ensemble de la logique opérationnelle d'un pilote automatique pour hélicoptère: la fonction de gestion des modes supérieurs de pilotage.
Cette cible est restreinte mais d'une complexité suffisamment représentative pour pouvoir apprécier les qualités des deux méthodes.

4. Evaluation de STATEMATE

4.1 Rappels sur la méthode de HAREL

Entre 1983 et 1985, David Harel a créé le concept de Statechart, puis une méthode de spécification complète portant son nom. Cette méthode est le prolongement naturel, formalisé et graphique, des méthodes d'analyse structurée appliquées aux problèmes temps réel.
Statemate en a été en 1987 la première et demeure à ce jour la seule implémentation.

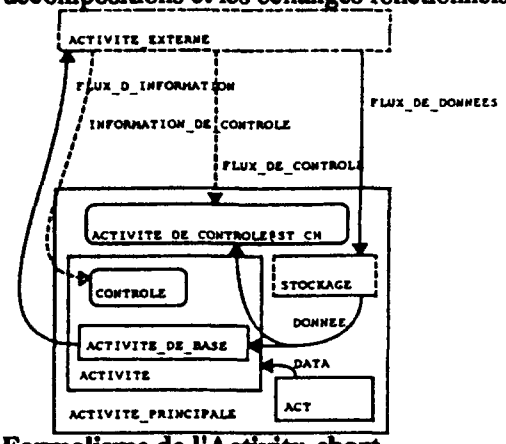
La méthode de Harel et l'outil Statemate permettent de définir un système selon trois types de vues :
fonctionnelle (activity-charts) et comportementale (statecharts) : c'est la spécification proprement dite,
enfin structurelle (module-charts):c'est le début de la conception.

Ces vues sont définies comme des emboîtements non limités de boîtes (resp. activités, états, modules) reliées par des flèches, reflet précis de la hiérarchie du système.
Une boîte peut être elle-même décrite par un diagramme (chart): c'est la facilité box-is-chart (cette facilité permet l'encapsulation).
Les boîtes sont dotées de noms et les flèches de labels.
Sur les flèches (flux et transitions) peuvent être placés des connecteurs simples ou

multiples à sémantique précise (correspondance, composition, etc).

4.1.1. - Activity-chart

Ce type de vue permet de définir les décompositions et les échanges fonctionnels.



Formalisme de l'Activity-chart

Pour cela, il utilise :

.des boîtes (activités) :

-consommant ou mettant à disposition des informations :

*boîtes rectangulaires : les activités externes (pointillées) et internes (continues),

*boîtes arrondies : les activités de contrôle, dont chacune est liée à un statechart, qui ont pour rôle de connaître et définir le comportement de leurs soeurs (exclusivement) et d'arrêter leur mère,

-simples lieux de transit d'information : les stockages de données (boîtes mi-pointillées, mi-continues) ;

.des flèches (flux d'information) :

- mettant à disposition:

.des événements(fugitifs, disponibles à un instant et perdus aussitôt, même non consommés) ou des conditions et des données (persistantes).

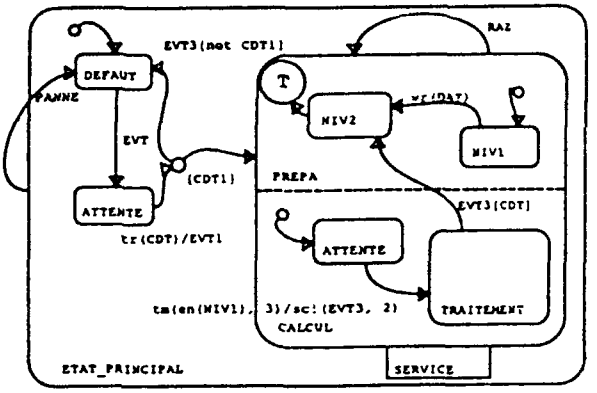
*de contrôle (flèches pointillées),

*de données ou mixtes (flèches continues).

Ces informations peuvent être structurées. Seuls les flux de contrôle correspondant à la connaissance et à la commande de l'état d'une activité par l'activité de contrôle sont implicites.

4.1.2. - Statechart

Un statechart définit le comportement hiérarchisé des fonctions.



Formalisme du Statechart

Pour cela, il utilise :

.des boîtes arrondies (états) ;

-un état peut contenir des états "ou" *dans lesquels le système ne peut se* trouver que successivement ;

-un état scindé par un ou plusieurs pointillés définit deux ou plusieurs états "et" dans lesquels le système peut se trouver à la fois ;

Ces notions peuvent se combiner au choix ce qui permet clarté et compacité; les états peuvent être dotés de réactions statiques produisant, sur déclenchement, des actions indépendantes de toute transition.

.des flèches (transitions) :
leur label (événement[condition]/action) comporte un déclencheur (événement[condition]) qui décide du passage (instantané) d'un état à l'autre et une action (action) qui est produite lors de ce passage.

Les événements et conditions peuvent être des combinaisons logiques d'événements ou conditions.

Les actions peuvent être des ensembles (non ordonnés) d'actions ; elles peuvent aussi être conditionnées par d'autres déclencheurs.

Les entrées par défaut et les connecteurs d'historique et de terminaison, complètent la sémantique des transitions.

Le temps n'intervient que de façon explicite et sous la forme :
- d'événements différés,
- d'actions programmées,
- de lien global entre pas (simulation, tests dynamiques) et unité de temps.

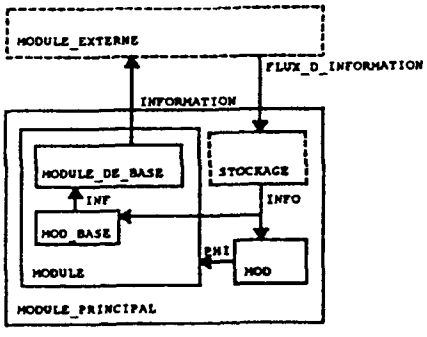
Les **liens entre statechart et activity-chart** sont opérés:

.par des événements
started (activity),
stopped (activity),etc.
par des conditions
active(activity), etc.
et par des actions
start ! (activity)
stop ! (activity),etc.)
placés :
-sur les labels des transitions,
-dans des réactions statiques des états;

.par des indications de corrélation entre états et activités
activity throughout state,
activity within state.

4.1.3. - Module-chart

Un module-chart permet l'allocation des fonctions et des flux.



Formalisme du Module Chart

Un module-chart ressemble à un activity-chart mais n'a pas de module de contrôle. Chaque module implémente une ou plusieurs activités, de sorte qu'il y ait un recouvrement complet de l'ensemble de celles-ci, à l'exception des activités de contrôle.

Les flux peuvent être regroupés (implémentation) différemment mais leur contenu est cohérent avec celui des flux des activity-charts.

Cette implémentation constitue le **lien entre module-chart et activity-chart**.

4.1.4. - Dictionnaire de données

Un dictionnaire de données rassemble les informations (et leurs relations) des diagrammes du modèle hiérarchisés par ces liens d'imbrication, de description ou d'implémentation. Chaque élément, graphique ou non, du modèle appartient à un diagramme. Son contenu sémantique est défini grâce à une grille spécifique.

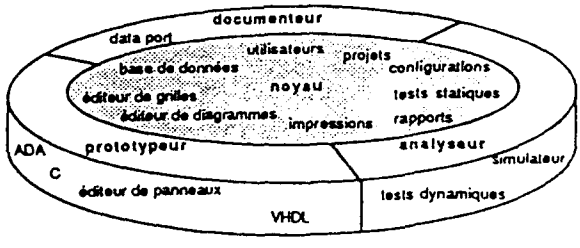
4.1.5. - L'outil Statemate

L'outil STATEMATE (V4.2) offre de nombreuses possibilités et spécificités.

Le modèle activity-charts statecharts est vérifiable statiquement (intra et inter-diagrammes). Il est de plus vérifiable dynamiquement par exploration systématique (utilisation des éléments, atteignabilité, impasse, conflit : compétition ou indétermination).

Il est exécutable directement (sans instrumentation supplémentaire) par simulation : l'utilisateur introduit les stimuli et en constate les effets sur l'animation graphique ou par l'examen des diverses informations, que ce soit en interactif ou en batch.
Les scénarios de test ou de simulation peuvent être enregistrés, rejoués et les traces des résultats conservées.

Un texte descriptif peut être associé à tout élément et en particulier aux activités (ou boîtes) élémentaires de la décomposition (texte, pseudo-code ou formalisme libre).



Structure de l'outil STATEMATE

Le modèle peut générer automatiquement du code C ou Ada; la structure du code des activités élémentaires est préparée mais laissée vide.
Un panneau graphique peut être défini, lui être connecté, et son code automatiquement généré.

Le tout peut être compilé avec le code éventuellement ajouté par l'utilisateur (dans les activités de base) et exécuté hors de l'environnement Statemate, constituant ainsi un prototype réaliste et interactif du futur système et de son interface.

4.2 Présentation des travaux

4.2.1. Modélisation

Les travaux de modélisation ont naturellement commencé par la définition précise de l'interface du système en vue de modéliser l'activity-chart de premier niveau, figeant ainsi la nature et le nombre des éléments intervenant dans la gestion de la logique opérationnelle.

Le premier souci a été de mettre rapidement en oeuvre une ébauche de modèle pour affiner la compréhension du comportement des modes supérieurs. Cette démarche a abouti à l'élaboration de plusieurs modèles ayant des approches différentes du problème.

Trois modèles ont été successivement construits. La puissance et l'ergonomie de STATEMATE a grandement facilité le travail de reprise des modèles (transfert d'éléments d'un diagramme à l'autre, gestionnaire des configurations du modèle intégré).

Le processus de validation statique du modèle ne fait pas à proprement parler de la modélisation.

Toutefois, dans la pratique, il est totalement intégré à la phase de modélisation car les tests de base peuvent être lancés à tout moment.

Ils sont particulièrement utiles car ils viennent en complément des vérifications de cohérence syntaxique et sémantique effectuées automatiquement lors de la saisie des informations.

La vérification statique se compose de tests de:

- **rectitude** : boucles de transitions, utilisation des informations conforme au type...
- **complétude** : transition sans label, événement défini et non utilisé...

4.2.2. Simulation

La modélisation étant intrinsèquement formelle, le modèle est exécutable sans aucune instrumentation et dans la pratique, la phase de simulation intervient dès qu'une ébauche de modèle validé statiquement est disponible (la validation statique est facultative mais recommandée).

La simulation interactive constitue de ce fait l'outil privilégié de mise au point du modèle. Dès qu'une anomalie de fonctionnement est mise en évidence par la simulation, la correction adéquate peut être apportée au modele.

Par un processus itératif, le modèle est progressivement mis au point. Il est à noter que la simulation peut avoir pour objet un sous-ensemble du modèle (dont la mise au point est particulièrement délicate, par exemple).

La simulation interactive se déroule pas à pas, mettant en évidence les mécanismes mis en jeu pour aller d'un état à un autre.
Il est également possible d'enchaîner automatiquement les pas de simulation en "super-pas" pour aller d'état stable en état stable.

L'interface de la simulation interactive est réalisée à l'aide de fiches à renseigner présentant la liste des événements et des conditions intervenant dans le modèle.

STATEMATE aide l'opérateur en indiquant quelles informations peuvent avoir une incidence sur le modèle au prochain pas de simulation.
Cette interface ne présente pas la convivialité d'une interface graphique. Son utilisation est lourde, surtout pour la simulation d'un modèle complet.

La simulation peut également être utilisée en mode batch en exécutant des fichiers de commandes générés manuellement ou par un enregistrement en simulation interactive.

4.2.3. Prototypage

Le prototypeur est un outil dont la fonction est de générer un code (C ou ADA) à partir du

modèle de spécification. Ce code est totalement détaché de l'outil qui l'a généré et peut donc être porté sur une autre station de travail dépourvue de l'outil STATEMATE.
La structure du code des activités (activity-charts) élémentaires est préparée, l'utilisateur peut éventuellement y mettre un code correspondant à la fonction.

Une option *debug* permet au prototypeur de créer un lien entre le code généré et les entités de STATEMATE, les services rendus sont les mêmes que ceux offerts par un debugger classique (points d'arrêt, observation et modification d'entités de STATEMATE, traces).

Enfin, un panneau graphique d'interface peut être connecté au code généré par le prototypeur.

L'ergonomie et le réalisme du prototype a permis son exploitation par une équipe d'ingénieurs spécialistes du pilotage automatique des hélicoptères. Chacun d'eux a appliqué les scénarios de tests qui lui semblaient pertinents pour s'assurer que le prototype fonctionne correctement et ainsi identifier rapidement les anomalies du modèle.
Cette constatation démontre l'intérêt du prototypage comme instrument de validation de la spécification; il offre un rapport:
nombre d'erreurs détectées/temps de détection
particulièrement élevé.

4.2.4. Validation dynamique

La validation dynamique se présente, d'une certaine manière, comme le complément du prototypage et de la simulation. Elle utilise des méthodes systématiques là où le prototypage fait appel à l'intuition.
Elle tente de répondre aux questions du type "Est-il possible que le modèle...?" plutôt qu'à celles du genre "Que fait le modèle lorsque...?".
Pour ce faire, l'outil de test dynamique de STATEMATE possède des mécanismes internes de génération de scénarios de tests alors qu'en simulation, ces scénarios sont bâtis par un opérateur.

L'outil de test dynamique permet de tester:
- la non-atteignabilité d'états interdits.
- l'existence d'impasse.
- la présence d'indéterminismes du modèle.
- l'utilisation des éléments du modèle.
- la présence de conflits d'affectation de valeurs à des paramètres.

En théorie, la puissance de cet outil est séduisante et il est tentant de l'appliquer sans retenue au modèle complet.

Une tentative de vérification complète des règles d'incompatibilité des modes a été réalisée. Après 48 heures de CPU (sur SUN SPARC1) la vérification n'était toujours pas terminée. Il a fallu se rendre à l'évidence que ce genre d'outil devait être manipulé avec discernement si on veut en tirer des résultats. Il est impératif, sur des modèles compliqués de limiter le champ d'investigation.

En pratique, il a été **démontré** que les états interdits (par les règles d'incompatibilité) ne pouvaient être atteints sous les hypothèses restrictives suivantes:
- les capteurs demeurent valides
- les trims sont embrayés
- les directeurs de vol ne sont pas engagés.

Ce résultat partiel a été obtenu en 23 heures CPU.

Une démarche complémentaire a consisté à introduire sciemment une erreur dans le modèle (une transition incorrecte conduisant à une combinaison de modes interdite) et à vérifier que l'outil était capable de la retrouver. La détection a pris 8 minutes CPU, le scénario trouvé par l'outil a été vérifié a posteriori en simulation interactive.

La validation dynamique possède donc des atouts qu'il serait dommage de ne pas exploiter.

Il est raisonnable de penser qu'en prenant certaines précautions dans la conception du modèle, la validation dynamique puisse être facilitée (cette contrainte n'a pas été prise en compte dans notre modèle).

L'accroissement probable de la puissance CPU dans les années futures milite également en faveur de l'approche "tests dynamiques".

4.3 Apports de la méthode

La méthode de HAREL présente une grande homogénéité au niveau des concepts manipulés;
les statecharts, originalité de la méthode, s'intègrent naturellement aux activity-charts pour obtenir une approche très visuelle du problème de la spécification.

Les principes mis en oeuvre, s'ils sont novateurs, s'appuient sur des notions largement répandues dans l'industrie (diagrammes états/transitions, diagrammes de flots de données). Ils permettent néanmoins de construire une spécification formelle du système.

La méthode de HAREL est supportée par l'outil STATEMATE qui permet de tirer profit de la puissance de la méthode.

STATEMATE, outre ses fonctions d'édition et de vérification statique des éléments de la spécification, permet d'exécuter le modèle de spécification.

L'intérêt de la simulation n'est plus à démontrer; **pour les systèmes complexes, l'obtention d'une spécification cohérente et validée impose quasiment d'y avoir recours.**

STATEMATE offre une panoplie d'outils intégrés qui ont pu être appréciés au cours de l'étude:

- **une base de données** (gérant les éléments de la spécification) autour de laquelle s'articulent des utilitaires d'interrogation et de gestion de configuration.

- **un prototypeur** générant un code détaché de l'outil et donc portable.

- **un outil de validation dynamique** générant automatiquement des scénarios de test.

Des améliorations de STATEMATE sont attendues :
-correction des erreurs du prototypeur.
-connexions de panneaux d'interfaces graphiques à la simulation, en vue de la doter d'une bonne ergonomie.

STATEMATE s'est révélé être particulièrement bien adapté à la spécification de la logique d'engagement des modes supérieurs.

Lors de cette étude, les limites de la méthode et de l'outil ne sont pas apparues (sauf peut-être au niveau de la validation dynamique). Il est donc permis de penser que la méthode de HAREL et STATEMATE sont bien adaptés à la spécification du comportement des systèmes tels que les pilotes automatiques d'hélicoptère.

5. Evaluation de G2

5.1 Rappel sur la méthode

La méthode supportée par G2 s'articule autour de la constitution d'une base de connaissances construite à l'aide de l'outil G2.

Une base de connaissances est principalement constituée :

-d'objets

-de tables d'attributs, qui permettent de décrire les caractéristiques de chacun des objets.

-de classes d'objets organisées suivant une structure hiérarchique.

-de variables et de paramètres : objets dont les valeurs sont des nombres, des symboles ou du texte.

-de connexions et de relations qui représentent les relations physiques, logiques ou temporelles existant entre les éléments de la base de connaissances.

-de règles qui définissent le comportement du système

-de procédures : ensemble de commandes du langage procédural de G2.

-de fonctions permettant d'effectuer des opérations arithmétiques.

La spécification se présente comme une modélisation du système, réalisée à base d'objets et de règles.

Les objets décrivent les éléments de la spécification d'ordre structurel. Tout objet est une instanciation de classe.

Les classes d'objets sont organisées de manière hiérarchique de façon à ce que les sous-classes puissent hériter des attributs des classes supérieures auxquelles elles appartiennent. Réciproquement, la hiérarchisation des classes permet de définir les attributs communs à plusieurs objets au niveau de la classe supérieure à laquelle les objets en question appartiennent.

Les règles décrivent l'aspect comportemental du modèle. Elles renferment en fait la connaissance de l'expert, dans la mesure où elles déterminent comment le modèle doit répondre aux diverses sollicitations.

Elles opèrent essentiellement sur les attributs des objets et peuvent donc, de ce fait, être génériques.

Une règle est composée de deux parties :

-l'antécédent, qui décrit les conditions d'activation de la règle.

-la conséquence, qui décrit ce qui est effectué lorsque la règle est activée.

Un moteur d'inférence permet l'invocation des règles suivants différents mécanismes :

-Chaînage avant: la règle est invoquée lorsque les paramètres figurant dans l'antécédent reçoivent une nouvelle valeur.

-Chaînage arrière: une règle R1 est invoquée lorsque sa conséquence opère sur un paramètre dont le moteur d'inférence a besoin pour évaluer l'antécédent d'une règle R2 qu'il doit invoquer.

-Invocation de règles par catégorie.

-Invocation périodique de règles.

-Invocation des règles associées à un objet particulier.

Ce moteur d'inférence est dit "temps réel", ce qui signifie en fait qu'un temps de réponse de l'ordre de la seconde est garanti.

L'outil G2 est disponible sur la plupart des stations de travail du marché

5.2 Présentation des travaux

5.2.1. Modélisation

La phase d'analyse a conduit à l'identification des principes généraux gérant les modes supérieurs et en particulier la notion de compatibilité entre modes.

Ces principes peuvent être décrits par un nombre limité de règles de comportement.

La modélisation "objet" est réalisée à partir des divers éléments de spécification intervenant dans les règles de comportement: les axes de pilotage, les directeurs de vol, les modes supérieurs et le sélecteur de navigation.

Les classes d'objets ont donc été définies et hiérarchisées

A chacun de ces objets ont été associés des attributs et les valeurs de ces attributs définissent les modes dont les engagement/armement sont autorisés sur l'axe.

Les différents modes supérieurs sont des instances d'une de ces classes. La nature et le nombre de leurs attributs sont fonction de la classe à laquelle ils appartiennent, ce qui définit en fait un type de comportement.

La représentation "objet" résout intrinsèquement une partie du problème de la vérification dans la mesure où chaque objet respecte le formalisme de la classe à laquelle il appartient, ce qui garantit un premier niveau de cohérence.

L'écriture des règles servant à modéliser le comportement du système, intervient logiquement après la création des objets. Un certain nombre de variables (validités, état trim, captures) complètent la représentation objet. Une relation *incompatible* est utilisée pour aider à décrire les incompatibilités entre modes.

Comme exemple de règle générique opérant sur une classe, on peut présenter la règle qui désengage les modes actifs à l'engagement d'un mode incompatible.

> whenever the *etat* of any *mode m* receives a value
> and when the *etat* of m is *actif*
> and the *etat* of any *mode m2* that is *incompatible* m is *actif*
>
> then conclude that the *etat* of *m2* is *inactif*

Cette règle fait intervenir explicitement :
- la classe *mode* et son attribut *etat* avec ses valeurs *actif* et *inactif*
- la relation *incompatible*

Cette règle, très générique, s'applique pour chaque instance de la classe *mode*.

Cet exemple donne une idée de la lisibilité des règles. Les entités créées pour les besoins de la modélisation portent des noms français.
L'utilisation de termes anglais conjuguée à un choix judicieux (*incompatible-with* au lieu de *incompatible*) aurait permis d'obtenir une

formulation des règles proche du langage naturel (de l'anglais!).

Le moteur d'inférence de G2 est doté de nombreuses possibilités d'invocation de règle; seuls les chaînage avant et arrière ont été utilisés pour chaque règle.

Remarque

Une limitation de la méthode liée à l'utilisation de l'outil G2 apparait à ce stade des travaux. L'ensemble des modes supérieurs a été partitionné en un ensemble de classes. Chaque mode appartient donc à une classe et une seule. Comme une classe définit un type de comportement, il est donc nécessaire de réaliser une partition fine de l'ensemble des modes pour traduire la diversité des comportements. Le nombre d'instances d'une classe terminale se trouve de ce fait bien souvent réduit à un ou deux.
Dans ce type d'approche, seule la hiérarchisation des classes permet une réduction de la complexité par une fédération des comportements élémentaires.

Une approche permettant de construire plusieurs partitions de l'ensembles des modes aurait permis d'obtenir des classes de cardinal plus élevé.
Chaque mode aurait alors appartenu simultanément à plusieurs classes d'objets, chacune d'entre elles reflétant une facette de son comportement.
Cette approche aurait nécessité l'utilisation de la notion d'héritage multiple, que l'outil G2 ne possède pas.

5.2.2. Simulation

Le modèle obtenu est exécutable, il se prête donc à la simulation.
L'étape suivante des travaux a consisté en la réalisation d'une interface de simulation.
Une modélisation graphique des éléments intervenant dans cette interface a été constituée. Cette interface a été réalisée sans véritable soucis d'obtenir une ergonomie voisine de celle du système réel. Sa fonction est uniquement l'alimentation du modèle en données et la visualisation des états obtenus. Sa construction a largement fait appel aux composants d'interface disponibles dans les librairies complémentaires de G2 (objets graphiques permettant de lancer des actions G2, d'affecter des valeurs à des variables symboliques...).

Les simulations "en batch" avec exécution automatique d'un fichier de commandes-opérateurs datées, sont également possibles.
La simulation s'exécute alors suivant le séquencement décrit dans le fichier de commandes; les états du modèle sont enregistrés dans un fichier de traces et sont simultanément visualisés sur l'interface de simulation.

Les simulations interactives et batch utilisent des concepts voisins et sont fortement intégrées. La simulation interactive peut être utilisée pour générer le fichier de commandes de la simulation batch (par enregistrement daté des commandes); les fichiers de résultats sont réalisés par le même utilitaire.

La phase de simulation est naturellement fortement couplée avec la phase de modélisation.

La mise au point du modèle est un processus récurent; la simulation permet de confronter le modèle à la spécification textuelle, de diagnostiquer rapidement les erreurs par des simulations interactives.

5.2.3. Validation

Les contrôles internes à G2 assurent la cohérence syntaxique du modèle et permettent d'éviter la phase de validation statique.

La validation du modèle consiste donc en une validation dynamique réalisée avec l'environnement de simulation.

La mesure de la couverture du modèle est réalisée à l'aide d'un utilitaire marquant les situations rencontrées au cours des simulations batch.

En pratique il a été possible de réaliser la couverture des points suivants de la spécification :
- -atteignabilité des états d'engagement (d'armement le cas échéant) en couplage et en directeur de vol pour chacun des modes.
- -atteignabilité de chacun des états systèmes.
- -vérification que les états systèmes non permis (modes incompatibles) ne sont pas rencontrés lors des simulations.

Cet utilitaire fournit la métrique nécessaire à la construction des fichiers de simulation batch qui constituent les tests de couverture du modèle. La construction de ces fichiers peut

être effectuée soit par une simulation interactive, soit par une édition directe.

Cette couverture est relativement limitée; si l'approche objet retenue intègre la notion d'état système, elle ne formalise pas celle de transition d'état. Le changement d'état est effectué par l'intermédiaire des règles de comportement.

La couverture des transitions d'état ne peut donc pas être effectuée avec le formalisme du modèle.

Pour pallier ce manque, l'approche adoptée a consisté à générer automatiquement un automate à états finis à partir du modèle.

La simulation est utilisée pour générer les transitions entre états: pour chacun des états du système, toutes les actions possibles sur les éléments de l'interface sont simulées; les transitions vers les états système atteints sont alors créées.

Pour obtenir un jeu de tests exhaustif, il reste à parcourir l'automate en passant par toutes les transitions d'état.

L'algorithme retenu est élémentaire: A partir d'un état initial, une transition non parcourue est franchie, ce processus se répète jusqu'à ce qu'une impasse soit atteinte (une impasse pour cet algorithme est un état au départ duquel toutes les transitions ont été parcourues), ce parcours correspond à un test.

Un nouvel état initial est ensuite choisi parmi les états possédant encore des transitions sortantes non parcourues.

Le processus décrit plus haut est reconduit. L'algorithme est arrêté lorsque, pour chaque état, toutes les transitions sortantes sont parcourues.

Cette approche a été maquettée en limitant le vecteur d'état du système aux modes engagés (pas d'armement de mode), en limitant l'interface aux boutons d'engagement des modes et avec le sélecteur de navigation dans la seule position NAV.
L'automate obtenu ne comporte que 26 états et 260 transitions et le nombre de tests assurant la couverture est de 30.

Les études sur la génération automatique de tests ont démontré la faisabilité de l'approche automate à états finis pour valider le modèle.

L'application de cette méthode au modèle complet risque cependant d'en révéler les limites, notamment lorsque le nombre de composantes du vecteur d'état s'accroit (modes armés, directeur de vol) et que tous les éléments de l'interface du modèle sont pris en compte.

5.2.4.Optimisation du modèle

Le modèle présenté dans le chapitre précédant possède une structure voisine de celle du document de spécification textuel qui a servi de base à la modélisation. Les aspects comportementaux du pilote automatique sont décrits fonction par fonction.

La démarche qui a prévalu lors de la rédaction des règles de comportements, a consisté à privilégier le caractère systématique au dépend de la généricité.

Les principes généraux régissant le comportement des modes transparaissent pourtant au travers de l'aspect systématique de certaines règles. C'est à ce niveau que se situe véritablement l'expertise et c'est cette approche que l'on a tenté de promouvoir en reprenant le modèle.

L'objectif est d'obtenir un modèle concis, constitué de règles que les modes doivent suivre plutôt que de règles formalisant le comportement de chacun des modes.

Le travail de reprise du modèle a consisté à reprendre les règles groupe par groupe.

L'éditeur syntaxique de règles s'est révélé être bien adapté à une utilisation par des personnes peu rompues à la sémantique du langage. Son ergonomie a été particulièrement appréciée; il propose à chaque caractère saisi la liste des mots autorisés, vérifie en ligne la cohérence de la règle éditée. L'éditeur autorise l'existence d'états intermédiaires syntaxiquement incohérents, l'intérêt pratique de cette facilité est évident lors d'une modification de règle.

A chaque étape de la modification du modèle, le recours systématique à la simulation a permis de s'assurer que le comportement du modèle demeurait inchangé.

Le nouveau modèle a beaucoup gagné en généricité, en concision et en lisibilité. La maintenabilité du modèle a progressé.

A titre d'exemple, l'adjonction d'un nouveau mode de pilotage se limiterait à la création d'un nouvel objet par instanciation de la classe auquel le nouveau mode appartiendrait et au renseignement de quelques attributs (sous réserve que ce nouveau mode ait un comportement apparenté à un comportement déjà rencontré).

La démarche adoptée dans cette phase des travaux, pourrait certainement être encore poursuivie. Des gains en matière de lisibilité des règles pourraient facilement être obtenus par un choix plus adéquat des noms d'attributs, de relations et de procédures.

Toutefois, les limites de G2 en matière d'héritage multiple (décrites dans le paragraphe précédant) se sont fait fortement ressentir et constituent l'obstacle principal au gain d'un ordre de grandeur en matière de généricité du modèle.

5.3 Apports de la méthode

La méthode basée sur G2 s'articule autour des notion de représentation objet et de système expert.

Ces techniques sont très novatrices, surtout dans le domaine de la spécification. Leur utilisation combinée a conduit à l'obtention de résultats satisfaisants; la gestion de la logique d'engagement des modes supérieurs et la visualisation associée ont pu être modélisées à l'aide de la méthode.

Le modèle obtenu est exécutable et son interface graphique possède de réelles qualités d'ergonomie qui rendent son exploitation aisée par des opérateurs n'ayant aucune connaissance des techniques informatiques mises en oeuvre pour construire le modèle.

Cette qualité pourrait être avantageusement mise à profit pour constituer les scénarios de validation.

La maintenabilité du modèle obtenu est bonne. De plus, la formation requise pour effectuer la maintenance est beaucoup plus réduite que le caractère novateur des techniques utilisées aurait pu le laisser supposer.

L'approche "objet" est en fait assez intuitive, l'éditeur de règles de G2 allie la puissance au caractère didactique. La preuve en a été donnée par l'usage: des modifications importantes du modèle ont pu être réalisées rapidement avec un taux d'erreurs très faible.

L'apport de la méthode dans la phase de spécification est démontré, toutefois il est difficile d'envisager une extension importante de son champ d'opération en dehors des aspects gestion de la logique opérationnelle.

La méthode basée sur G2 apparait plutôt comme une méthode de prototypage particulièrement rapide de l'aspect comportemental des systèmes.

L'environnement de simulation permet la validation de la maquette et les scénarios de tests de validation sont réutilisables en phase de validation du système réel.

La puissance de l'approche orientée objet a été appréciée et fait d'autant plus regretter l'absence de la notion d'héritage multiple qui aurait permis à la méthode de gagner en concision et rapidité de construction du modèle.

6. Conclusions

Ces techniques ont permis de construire des spécifications et de les valider avec des degrés de complétude plus ou moins importants.

L'intérêt de ces techniques a été clairement démontré et elles apparaissent comme des éléments-clés dans une démarche organisée de réduction du coût de développement des logiciels des systèmes complexes.

Il reste désormais à montrer qu'une spécification peut être entièrement formalisée par l'utilisation d'une méthode de description des aspects comportementaux couplée à une méthode de description des aspects transformationnels, et que ce couplage est industriellement utilisable et bénéfique.

La mise en pratique sur un projet d'envergure nous permettra de confirmer le bien fondé de nos suppositions sur la maîtrise des coûts et d'observer les réactions de nos clients vis-à-vis de ces nouvelles façons de travailler.

Déjà la définition d'une méthodologie intégrant ces nouvelles techniques est envisagée et celle-ci sera présentée aux Services Officiels pour obtenir leur avis quand a son application dans les systèmes développés pour l'aéronautique, tant civile que militaire.

Références

"Comparative Evaluations of Four Specification Methods for Real-Time Systems"
Carnegie-Mellon university Technical report (CMU/SEI-89-TR-36 ESD-89-TR-47)
(D.P.WOOD,W.G.WOOD)

"Les langages synchrones: des logiciels pour la spécification et la conception des systèmes temps réel de traitement de l'information"
C2A 31 mai 1989 Rapport n°1bis
(N.BENVENISTE)

"STATEMATE: A working environment for the development of complex reactive systems"
IEEE Trans. Soft. Eng. Vol 16 N°4 Apr 1990 pp403-413
(D.HAREL)

"Pilotage automatique d'hélicoptère: des spécifications enfin validées"
REAL-TIME SYSTEMS: (acte des conférences) 12-15 Janvier 1993
(F.DELHAYE, C.JOUBERT)

DESIGNING AND MAINTAINING DECISION-MAKING PROCESSES

Andrew Borden
SHAPE Technical Centre
P.O. Box 174
2501 CD
The Hague
The Netherlands

1. SUMMARY

The design of the optimal strategy for decision making is intractably difficult when information sources (sensors) provide different amounts of information and have different costs associated with their use. This paper describes a sub-optimal design algorithm which can be further simplified if the user specifies some parts of the decision process to be executed at run-time. The design algorithm is provided with diagnostics so that the result can be compared with accuracy and response time requirements, and a learning module which enables new information to be incorporated into the knowledge base. The complexity of this algorithm is compared to that of the optimal design algorithm.

2. INTRODUCTION

There are many interesting applications for real-time Decision Making Processes (DMP's). Examples are Electronic Support Measures (ESM) logic design, Electronic Countermeasures (ECM) optimization, Indications and Warning logic design and fault isolation/diagnosis strategy in avionics. DMP's must do the following:

- Apply a set of sensors to determine the characteristics of an object in the environment. (In this context, a "sensor" is any instrument which observes an object and reports the value of an attribute. The instrument could be, for example, an interferometric frequency measurement device or a human being counting objects in the environment.)
- Compare the sensor readings to a knowledge base
- Provide an identification or classification of the object despite residual uncertainty

...subject to response time and probability of error constraints. The task is made more difficult by the fact that the sensors have varying penalties associated with their use. For example, it may take much longer for a Radar Warning Receiver (RWR) to determine the scan rate of a radar than to determine the pulse recurrence interval. The difference in response time could mean an unacceptable delay in the initiation of countermeasures against the radar. It is therefore important, to select an efficient strategy for the order in which parameters are considered to obtain the maximum information with the least cost.

Finding the optimal DMP strategy however, is a Non-Deterministic, Polynomial time (NP), complete task. (1) This means that the algorithm to solve the problem may be simple to write, but not feasible to run because of the great computational requirements. It is necessary therefore, to accept a sub-optimal, but "good enough", result. Moreover, the control structure of any logically correct, run-time procedure is so complex that standard methods of structured analysis for software design are not applicable. The result is that DMP's may be designed in an unstructured way and that performance (response time and/or confidence) requirements can be exceptionally difficult to satisfy.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

A related, potential difficulty is caused by the fact that the knowledge base for a DMP is subject to change. For example, a new sensor could be installed in a part of the spectrum not covered before and its contribution to situation assessment must be taken into account. If the original DMP design was produced by unstructured methods, the update task can be as difficult as the original design task.

The task of specifying a DMP and the possibility of developing CASE tools for design is discussed by Borden and Cinar in Reference 2. The implications for NP-Completeness of Radar Warning Receiver (RWR) design are discussed in Reference 3.

The purpose of this paper is to describe an algorithm which can be used to design an efficient decision making strategy for a real-time DMP. The procedure to be described is a greedy algorithm which selects the sensor providing the greatest ratio of new information to cost at each node of the decision tree. It will not however, guarantee a globally optimized decision strategy.

3. RELATED RESULTS

Goodman and Smyth (4) describe an information theoretic algorithm used to build decision trees. There are two important differences between the Goodman/Smyth Algorithm (GSA) and the algorithm described in this paper (DMP Designer):

- The GSA assumes that every sensor has the same cost whereas, the DMP Designer assigns different costs (response times) to different sensors.
- The GSA selects a new sensor to expand a node based on the amount of mutual information between the new sensor and the environment. The DMP Designer selects a new sensor based on the reduction in conditional entropy at each node, taking into account the information that other sensors have already provided. Unless the overlap in sensor readings for different objects in the environment is very uniform, the DMP Designer will almost certainly provide a different and more efficient result.

Goodman and Smyth do not discuss diagnostics which could be made available to the user. However, they do describe a learning module which can compute statistics from observations of a new object and update the knowledge base accordingly. This learning module has been adapted for use in the DMP Designer.

4. DESCRIPTION OF THE ALGORITHM

The inputs to the procedure include a description of the sensor suite:

- The number of sensors
- The response time (or other cost) of each of the sensors
- A partition of the set of possible readings for each sensor (j) into "windows" (k) or discrete values

The user also provides a knowledge base which includes the a priori probability of occurrence of an object of type i (O_i) in the environment and the conditional probability that the sensor j will report a value in window k when observing an object of type i: $P(W_{jk}|O_i)$.

Finally, the user provides a stopping rule to ensure that a branch of the decision tree terminates when the probability of an incorrect classification is small enough.

The first step in the algorithm is to find a branch of the current decision tree which does not have a terminal indicator. The existing tree could be the root node if the tree is being built from scratch, a branch of a partial tree provided by the user or a branch of the tree as it has been constructed by the algorithm thusfar. If every branch has a terminal indicator, the algorithm prints the

completed tree and the user may proceed to the diagnostics module.

Once the branch has been selected, the sensors not yet used on that branch are retrieved. If the number of sensors already used equals the total number of sensors, a failure notification is appended to the branch and the algorithm goes back to step one to find another branch. Otherwise, the available sensor providing the maximum payoff in bits of independent information per second is selected. The node at the end of the branch is expanded for all possible (discretized) attribute values which the new sensor might provide.

Finally, the algorithm tests all the new nodes which have been generated. If the stopping rule is met, a terminal indicator is appended in the form of a positive classification result. When this activity is completed, the algorithm returns to the first step.

The diagnostics module reports the confidence level for every positive classification, the a priori probability of every branch of the tree which terminates in failure, the latency time for every branch of the tree, the mean latency time and the expected throughput in bits per second. To correct classification failures which have a significant probability of occurrence, the user can specify a result and annotate the branch appropriately. If the latency time on any branch is unsatisfactory, he may terminate the branch earlier and and specify a result. After editing, the user may re-run the diagnostics to determine the performance of the modified tree.

This algorithm does not solve the NP-complete, global optimization problem. The problem it does solve however, is still very complex if the problem domain is large, i.e. if there are many different object types in the environment, many sensors and many windows for each sensor. To reduce the complexity of the design task, the user can specify a partial tree in advance. In this way, the user can circumscribe the work to be done by the algorithm and reduce its complexity. If the user places a terminal indicator on any branch, the algorithm will ignore all possible expansions at this node and work on the remainder of the tree. The ability of the user to specify part of the solution is especially useful if a change to the knowledge base occurs which affects only part of the existing decision tree.
Typically, the first sensor to be consulted is pre-determined either because it is a direct output of pre-processing or because it is an alarm which initiates the decision making activity. It is not necessary however, to specify any part of the decision tree in advance.

The learning module adapted from the GSA can be used to add to the knowledge base by statistical analysis of a sample data set derived from a new object in the environment. It could (for example) add the parameters of a new radar emitter or wartime mode to the knowledge base used by an ESM system. This module automatically computes the statistics of the parameter distributions, adds them to the knowledge base and redesigns the decision tree as required. It is necessary however, for the user to provide the a priori probability of occurrence of the new object.

5. INFORMATION THEORETIC CONSIDERATIONS

The fundamental consideration in designing this algorithm is to obtain the maximum amount of mutual information between the selected sensors and the domain with the least cost (usually time). At each node of the decision tree, the maximum payoff sensor (in terms of bits of new information per second) is selected. The node is expanded for all the possible outcomes (windows) for this new sensor. The probability of correct classification at each new

node is then computed and terminal indicators are added to each finished branch.

As part of the initialization, the algorithm computes the a priori probability that a sensor j will report a value in window k:

$$P(W_{jk}) = \Sigma_i\ P(O_i) * P(W_{jk}|O_i) \quad (1)$$

This result is used to compute the probability that the object being observed is of type i, given that sensor j has reported a reading in window k:

$$P(O_i|W_{jk}) = (P(O_i) * P(W_{jk}|O_i))/P(W_{jk}) \quad (2)$$

The next step is to compute the probability that the object is of type i given a sequence of windows for the sensors which have already een used. If $\{W_{jk}\}$ is a list of windows (k) for sensors (j), then:

$$P(O_i|\{W_{jk}\}) = (P(O_i) * \prod_{jk} P(W_{jk}|O_i))/P\{W_{jk}\} \quad (3)$$

This equation is true because the sensors are conditionally independent. If pairwise (or n-wise) relationships exist between the sensors for a given object-class, these clusters are identified as distinct objects or variants of objects in the environment.

If the conditional probability in Equation 3 exceeds the required confidence level for some i, this branch stops and a terminal indicator consisting of a classification is added. If not, all the unused sensors are evaluated to determine which will provide the greatest payoff in new information per unit cost. To do this, we use the formula for the remaining uncertainty about the environment given the windows $\{W_{jk}\}$ which have been reported to date:

$$H(O|\{W_{jk}\}) = \Sigma_i\ P(O_i|\{W_{jk}\}) * \mathrm{Log}_2 P(O_i|\{W_{jk}\}) \quad (4)$$

and the formula for the uncertainty which will remain after using a new sensor (j^) with windows ($W_{j\hat{}k\hat{}}$):

$$H(O|\{W_{jk}\}\wedge(W_{j\hat{}k\hat{}})) = -\ \Sigma_i\ \Sigma_{k\hat{}}\ P(O_i\wedge(j\hat{},k\hat{})|\{W_{jk}\}) * \mathrm{Log}_2 P(O_i|\{W_{jk}\}\wedge(W_{j\hat{}k\hat{}})) \quad (5)$$

where $P(W_{j\hat{}k\hat{}})$ is the a priori probability of occurrence of window k^ for sensor j^:

$$P(W_{j\hat{}k\hat{}}) = \Sigma_i\ P(O_i) * P(W_{j\hat{}k\hat{}}|O_i) \quad (6)$$

The new sensor (j^) which maximizes the difference between formula 4 and formula 5, divided by the response time, is the one which contributes the most new information per unit cost, so this sensor is used to expand the node being worked. The evaluation of the new nodes proceeds depth first, from front to back until there is a solution at the end of every branch, or the process runs out of sensors and fails.

For a more detailed discussion of the information theoretic equations used in this program, see Reference 5.

6. PROGRAMMING INFORMATION AND COMPLEXITY ANALYSIS

This algorithm was coded in PC Scheme, a version of LISP produced by the Texas Instruments Company. The elements of Scheme syntax used for this program are reasonably standard for LISP.

The complexity (C) of the algorithm described in this paper is on the order of:

$$C = N(i) * N(j) * \prod_j N(k(j)) \quad (7)$$

where:

- o N(i) is the number of object classes in the environment
- o N(j) is the number of sensors
- o N(k(j)) is the number of windows (k) for sensor j.

Since all subsets of the possible sensors to be used on each branch must be considered in the globally optimal algorithm, the complexity (C^) is on the order of:

$$C^{\wedge} = N(i) * 2^{**N(j)} * \prod_j N(k(j)) \quad (8)$$

where the middle part of the expression is the number of subsets of a set of N(j) elements.

For example, if there are five object classes in the environment, five sensors and five windows for each sensor, the complexity of the algorithm in this paper is on the order of 78,125 whereas the optimal algorithm is 484,375.

If, as is probable, we are able to specify the first sensor to be used, the complexity is reduced to 12,500 and 48,875 respectively. If we specify branches of the decision tree which are already satisfactory, the design task can be further simplified.

7. PLANNED ENHANCEMENTS AND APPLICATIONS

The LISP version of the algorithm is regarded as an exploratory prototype. This language is a convenient prototyping tool for an algorithm of this type because LISP programs are inherently modular and recursion is very easy to implement. The program is currently being re-hosted in C++.

The next module being developed will provide a front end for the existing algorithm so that the design of decision logic will be a turnkey operation. The input will be a data base containing the distributions of the values of attributes of the objects in the environment. The NATO Emitter Data Base is an example. The new module will find overlaps in attribute values, create windows and compute all the conditional probabilities needed to complete the knowledge base.

As one of several exercises done with the algorithm, the author has used it to design Radar Warning Receiver (RWR) logic for an environment consisting of four radar threats. Parameters from the NATO Emitter Data Base were used. Simulated intercepts of a fifth threat were then introduced to exercise the learning module. The resulting decision logic was further tested in an RWR simulation driven by the STC Radar Environment Simulator (RES).

8. CONCLUSION

If sensors provide different amounts of information and have different costs associated with their use, the design of an optimal strategy for sequential constraint satisfaction using these sensors is an NP-Complete problem. This paper contains a description of a sub-optimal design algorithm which reduces the complexity of the problem. The algorithm also allows the user to specify part of the solution to reduce the complexity even further. This algorithm applies when the probability distributions of sensor readings are known for all the object classes in the environment, but when there is inherent ambiguity (overlap) between the distributions. It is useful whenever efficiency at run-time is important. The algorithm has been implemented in LISP. A comprehensive set of diagnostics and a learning module are provided.

9. REFERENCES

1. Genesereth, M.R. and Nilsson, N.J.; "Logical Foundations of Artificial Intelligence", Morgan Kaufmann Publishers, Inc., Los Altos, California, USA, 1987.

2. Borden, A.G. and Cinar, Unver; "Specifying Decision-Making Processes", Lecture Notes in Artificial Intelligence 604, F. Belli and F.J. Rademacher (Editors), Springer-Verlag, 1992, pp 139 - 142.

3. Borden, A.G.; "Salvaging the ARC", Journal of Electronic Defense, October, 1988, pp 71 - 120.

4. Goodman, Rodney M. and Smyth, Padhraic; "Decision Tree Design Using Information Theory", Journal of Knowledge Acquisition, 1990, 2, pp 1 - 19.

5. LaFrance, Pierre; "Fundamental Concepts in Communication", Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, USA, 1990.

Discussion

Question L. HOEBEL

Do you have any conjecture abou the effect of users on complexity for large scale problems, that is, with more radars/parameters, do users have more or less effect on software (operation) complexity?

Reply

The relationships between the "complexity" in the environment and the complexity of the algorithm is given in the paper. As the environment grows, the increase in computational complexity will outpace improvements in processing power/speed. The domain expert will have to do more.

EMBEDDED EXPERT SYSTEM : FROM THE MOCK-UP TO THE REAL WORLD

Frédéric CAGNACHE

Thomson-CSF/RCM
178, Boulevard Gabriel Péri
92242 Malakoff Cedex
FRANCE

SUMMARY

This paper is presenting a methodology for the integration of an expert system in an embedded real time software running the VRTX operating system. We define the "Expert Unit" concept and its life cycle referring to a real experimentation. The production of operational embedded expert system becomes a reality because of the use of efficient tools and an original methodology for its integration.

This paper is showing how software engineering techniques and methods (Life-Cycle definition, identification of generic configuration items) can help an embedded expert system development to be more efficient and controllable than by using the Boehm's spiral model. This Life-Cycle is compatible with military quality standards (GAM T17 and DOD 2167a).

An expert unit is a software component which may be integrated as other (conventional) components of the application but, as including knowledge, it should be developed with specific methods and tools. Its structure is also specific and includes 3 parts: Knowledge base, Data interfaces, Programming interface. This structure is using the "abstract objects" XIA/XRete tool's facility.

These development life-cycle and Expert Unit concept imply a new design approach of Knowledge-Based Systems. During the definition phase, the components of the application and their functions are defined. Some of them are identified as heuristic or decisional (knowledge based) functions in the application. Specific life-cycle is then applied to these functions. A KBS becomes a software where expert units which encapsulate knowledge, and classical components are integrated.

The work described next is a complement of the work done by Thomson-CSF/RCM to apply expert system techniques to the problem of radar identification. This work has been carried out within the Thomson-CSF XIA/XRete project.

INTRODUCTION

We use to assume that an expert system is basically different from a classical software. A conventional software is made up variables, functions and procedures. But an expert system is made up an inference engine which is a classical software and a knowledge base that contains inference rules given by the human expert, elicited and formalised by knowledge engineer and exploited by the inference engine.

It implies that a set of specific methods and tools have to be used in the development of a software which includes a knowledge-based function. Furthermore, this set is used on the whole software development. An application is a classical software or an expert system.

In real world, most of applications which we need to develop, contains functions which are typically algorithmic and others which are typically heuristic or decisional. Both of it have to work, communicate and interface each other. First technical solution consist in design of complex systems which inter couple classical software and expert systems.

Such a solution cannot be chosen for embedded military application development. Coupling is not enough efficient to respect operational requirements (execution time, memory space). To respect such requirements the expert system have to be deeply (completely) integrated. An expensive solution is to develop anymore the expert system in procedural language.

In 1984, new techniques of rule base total compilation happend. With these techniques, the integration of an expert system in software becomes possible [Fages86] [Fages 88]. The result of this work is the XIA/XRete tool, a real-time expert system workbench. XIA/XRete is composed of a rule base compiler which generates target procedural language (C, C++, LTR3 or LISP) from rule base written in rule language and of an inference engine.

This paper is introducing the methodological and design approach associated to this kind of tools. It preserves creativity during Mock-up phase, all the development products are re-used over the life-cycle and no constraint is imposed by the expert system on the application architecture.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

METHODOLOGY

Here is defined the "Expert Unit" concept and its Life-Cycle. It has to be compatible with classical software life-cycle [Boehm81] [GAMT17] [DOD2167]. It is also derived from the ESA's expert system life-cycle [ESA85]. And the productivity is increased by comparison with the spiral model [Boehm88].

Usually, an expert system is a complete software where all parts of it (algorithmic or heuristic) are designed as an expert system. In a software development process an expert system component impose an "hegemony" i.e. the other components have to adapt their external interfaces to be able to work with it. In this way, development process becomes more difficult and expensive.

The Expert Unit concept has been defined to control more efficiently embedded knowledge-based systems development. This objective has been reached by looking upon expert system just as another software component. So that software engineering techniques and methods may be applied on it. This is possible by designing the expert unit as made up three parts which are the three configuration items defining completely an expert unit:

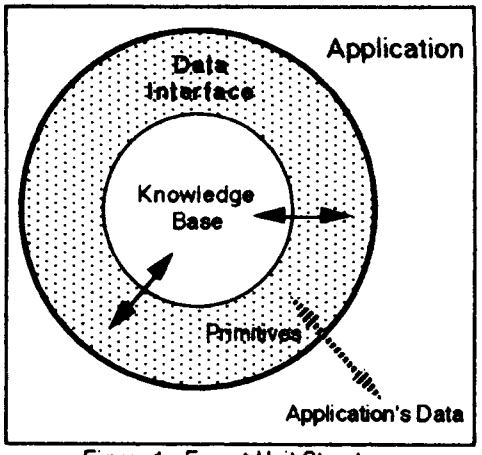


Figure 1 : Expert Unit Structure

The three configuration items are:

- knowledge base

  The knowledge base is the item where expert's knowledge is elicited in rule language. The knowledge base is the more specific item of the configuration because it contains no classical programming code. It gives the expert nature of the expert unit configuration.

  The applicable tools are the rule compiler then if needed the target language compiler. This is developed in the mock-up environment by a knowledge engineer.

  This is an element (referring to [DOD2167]) because considered as a compilation unit. The element concept needs to be adapted because this item may be compiled two times (once by rule base compiler and once by target language compiler).

  In the right way, the knowledge base must be completely developed at the end of the mock-up phase.

  It is possible, in a pragmatic way, to develop, in a first time, an incomplete knowledge base to integrate it in the application and to test it sooner with real data. In this case, the final and complete knowledge base could be obtained incrementally by successive changes. This way is not particularly useful because controlling the convergence of this succession of changes is very difficult.

- data interfaces

  Data interfaces are made up implementation directives of abstract objects (objects handled by knowledge base and physically located in the application's data) and interfacing primitives. Implementation directives use interfacing primitives to access to objects values.

  All data transfers between the knowledge base and the application must be done through data interfaces.

  In our experimentation, data interfaces use the abstract object facility of the XIA/XRete tool [XIA91]. This facility lets handle (abstract) objects in the knowledge base as the expert want to see them independently of their real implementation or location. So it lets specify formally for each object and its attributes how to calculate their value.

  The applicable tools are the rule base compiler (for implementation directives) and the target language compiler (for primitives). Data interfaces are produced, integrated and tested in the Mock-up environment by knowledge engineer for implementation directives and software engineer (from the application's team) for interfacing primitives.

  Data interfaces are a component (referring to [DOD2167]) because they are not a compilation unit. They are composed by two parts which are compiled by two different compilers.

Data interfaces are developed during the prototype phase. Changes of knowledge base bring to data interfaces changes.

- programming interface

  The programming interface contains all the function and procedure references between the expert unit and the application and the entry point of the expert unit (inference loop). It contains all the links (in or out) with the application in the call graph.

  Applicable tools are rule base compiler (to compile rule language primitives) and target language compiler. The programming interface is first developed by a knowledge engineer in the Mock-up environment to simulate the application working and rewritten in the application environment during the expert unit phase by a software engineer.

  Programming interface is an element (referring to [DOD2167]) as the knowledge base.

Below is described the Expert Unit Life-Cycle integrated into the application V Life-Cycle.

The Expert Unit Life-Cycle is made up three main phases, an optional phase (Knowledge Base porting) and two common phases with the application life cycle (Definition and Integration phases).

The three phases are:

- Mock-up phase

  During the Mock-up phase the expert knowledge is elicited and formalised by a knowledge engineer from human expert then the knowledge base is developed. Interfaces with the application are simulated.

  At the end of this phases the knowledge base must be validated. This is recommended but not absolutely necessary as we have seen it above (see knowledge base description section).

  All the developments are made in the mock-up environment (LISP). Application constraints are not considered in the mock-up development.

  Strong interactions exist between Definition and Mock-up phases because of the feasibility mock-up which is necessary in most of cases.

- Prototype phase

  During the prototype phase, the data interfaces are produced by considering the application interfaces. Data interfaces (implementation directives) are developed in the mock-up language. These are used as specifications to develop interfacing primitives in application development language. Then they are incrementally integrated to the mock-up which becomes in this way a prototype. After each primitive integration, no-regression tests are applied. So the prototype development is fully controlled and validated.

  The primitives are developed in a classical way (with software engineering techniques, tools and methods). The prototype is produced in the mock-up environment which is easier to use.

- Expert Unit phase

  During Expert Unit phase, the programming interface is rewritten in the application language.

  The expert unit is a set of source files which are compiled by the rule base and target language compiler and linked with the other application units in the embedded application environment.

The KB Porting phase may be necessary if another rule base compiler (with another rule language) is used in the application environment. But it is not necessary with the XIA/XRete compiler which generates code in both environments, mock-up (LISP) and application (C). This increases the development productivity and control.

The integration of the expert unit is made as for other units of the embedded application.

Changes are made from the prototype. Changes of the expert unit consist essentially of changes on knowledge base.

The complete Expert Unit life-cycle is shown in figure 2. We can see that "preliminary design", "detailed design" and "coding" phases are replaced by "Mock-up", "Prototype", (optional) "KB Porting" and "Expert Unit" phases. The expert unit is a software component which is pointed out as an expert function during the "Definition" phase.

Productivity is greatly increased because all the developments are re-used through the whole life-cycle from the mock-up (knowledge base) to the expert unit (data interfaces and programming interface). And the development process becomes also more controllable.

The productivity during changes is also increased because:

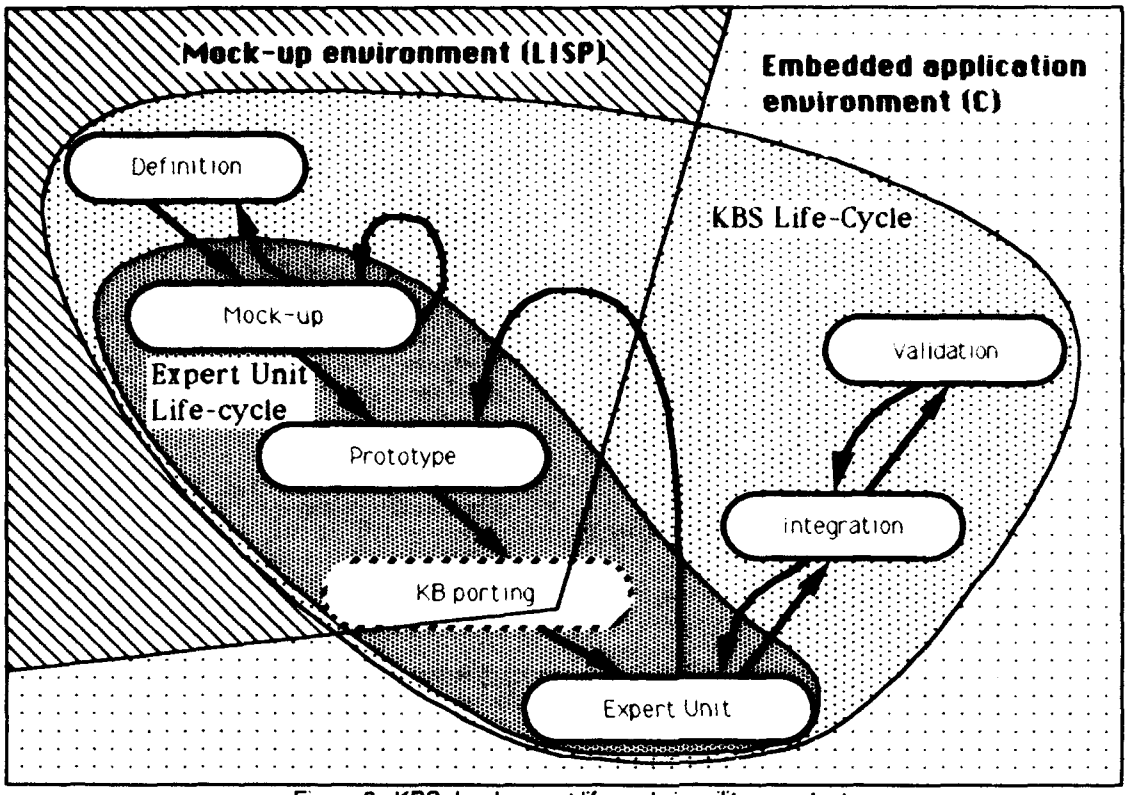


Figure 2: KBS development life-cycle in military context

- changes are made from the prototype in the mock-up environment which is more powerful and user-friendly
- changes are made essentially upon knowledge base which is easier to maintain because it is more modular and compact than its equivalent in procedural language.

I estimate that changes productivity increased about 5 to 7 times by using expert system techniques in complex functions of a software.

FROM A MOCK-UP TO AN EMBEDDED SOFTWARE

Here is shown how the methodology described above is concretely applied on an operational application.

To compare the performances and development efforts we develop two versions of the same function:

- one in a classical way in procedural language
- the second with expert system techniques

To apply the expert unit life-cycle, is to integrate the expert system mock-up of the radar identification function in a real-time embedded software. So that the mock-up is replacing the classical equivalent component of the embedded software.

The embedded software running the VRTX operating system masters a radar detector equipment. Constraints on memory space and execution time are very strong. Data structures are optimised in accordance to application's information processing. In any case, they are not adapted to expert system processing.

The mock-up is a radar identification demonstrator developed in a LISP environment on VAX workstation (for more details see [Galichet90], [Schang86], [Schang88a] and [Schang88b]). So memory and time constraints are very weak.

The mock-up knowledge base has been developed and structured as shown in figure 3: knowledge access its data only through data interface. So that physical implementation can change without bringing to changes on knowledge base.

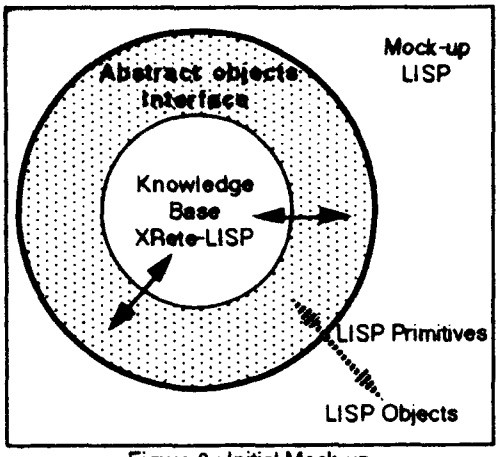


Figure 3 : Initial Mock-up

In a first time, C primitives are developed and unit tested. Then LISP primitives are replaced incrementally by their equivalent C primitives. At every replacement, non-regression automatic tests are applied to detect any error. This work permit to obtain the prototype which can access from LISP environment to application's data in their exact format through LISP-C interface. It guarantees also that the prototype is functionally identical to the mock-up (see [Cagnache92] for more details).

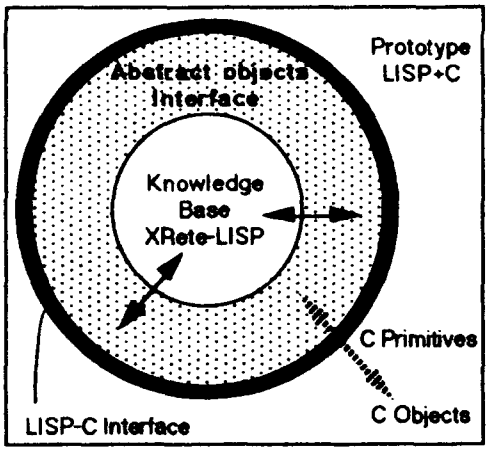


Figure 4 : Prototype

Because the XIA-C tool became available, we translated the knowledge into XIA-C syntax. This translation has been automatically made to ensure consistency between prototype and expert unit.

Then this set of XIA-C and C source files are compiled to obtain object files which can be linked with other object files of the application as anyone else.In this way, we obtain an executable program file that we can experiment and compare to the "classical" one.

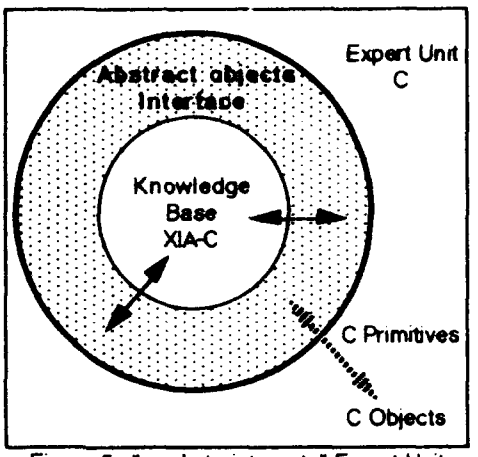


Figure 5 : "ready-to-integrate" Expert Unit

The performance of the expert unit presents the same difficulties on the same cases as classical unit. But the performances of the expert unit are worse than the classical one by a significant factor which is variable (see [Cagnache92] for more details).

This variability of the performance factor may be a mater to produce really operational expert units. I suppose that we will never can produce expert units which have same performances as classical one. Because the code of classical unit is optimised for one particular application but the expert unit code is generated by the rule base compiler which cannot be so efficient. The difference of performance is the cost of generic processing of knowledge management.

Some work is done to get better performance. We obtain some significant results:

- worst performance cases became better in a significant way
- inference engine has been optimised and this work gives the best performances improvement

So that we can say that we can use expert system techniques in reasonably time constrained software

CONCLUSION AND PERSPECTIVES

In this paper we have defined a KBS Life-Cycle to make its development and integration more controllable. This approach is based on the expert unit concept.

An expert unit is a software component which may be integrated as other (classical) components of the application but, as containing (encapsulating) knowledge, it should be developed with specific methods and tools.

Its structure is also specific and is made of 3 parts: Knowledge base, Data interfaces through which the knowledge base access to application's data, Programming interface through which the expert unit is called by the application and the expert unit calls other units of the application.

In the expert unit life-cycle, activities are clearly behaved. Mock-up phase addresses the knowledge base development, prototype phase the data interfaces development, expert unit phase the programming interface development. The KB porting phase is optional because not necessary if a tool like XIA/XRete is available (XIA/XRete works on both environments, mock-up and application). Then the expert unit is integrated with other components of the application in a classical way.

Changes are made from the prototype. All the developments are re-used over the whole life-cycle. So that development process productivity is increased.

This life-cycle is compatible with military quality standards GAM T17 and DOD 2167a. These development life-cycle and Expert Unit concept implies a new design approach. During the definition phase the components of the application and their functionality are defined. Some of them are identified as heuristic or decisional functionality (knowledge based) in the application. Then specific life-cycle are applied to them. A Knowledge Base System becomes a software where expert unit which encapsulate knowledge, and classical components are integrated.

Time performances of expert unit are worse but comparable (in a significant scale) to the performances of a classical equivalent unit. They become better by optimising knowledge base and inference engine. Some work is actually done to get them better.

This life-cycle needs to be completed by a knowledge base development method (from knowledge elicitation to Knowledge Base design and coding).

Quality standards need to be adapted to allowed integration of knowledge bases in embedded software.

Embedded intelligence becomes a reality.

REFERENCES

[AFNOR85] AFNOR : "Vocabulaire de la qualité du logiciel"
AFNOR ref. X50-109, July 1982

[Boehm81] B. W. BOEHM : "Software engineering economics"
Prentice-Hall, Englewood Cliffs, New Jersey 1981

[Boehm88] B. W. BOEHM : "A spiral model of software development and enhancement"
Computer, May 1988

[Cagnache92] F. Cagnache : "Embedded expert system: Methodology and experimentation"
12th international conference on artificial intelligence, June 1992, Avignon France

[DOD2167] DOD-STD-2167A : "Military standard defense system software development"

[ESA85] ESA : "Cycle de vie d'un système expert"
European Space Agency ref ESA/ESTEC/tms/85-446/MG/md, 1985

[Fages86] F. Fages : "On the proceduralization of rules in expert systems"; First France-Japan Symposium on artificial intelligence and computer science Tokyo, October 1986
in Programming of future generation computers, North-Holland,
Eds M. Nivat & K. Fuchi

[Fages88] F. Fages : "Rule based extension of programming languages : a proposal to integrate expert system into applications at the target language level"
8th International workshop on expert systems and their applications, May 1988, Avignon France

[Fox90] M. S. Fox : "AI and expert system : myths, legends and facts" IEEE Expert, February 1990

[Galichet90] Ph. Galichet & al : "Système expert temps réel pour l'identification de radars" ; 10th international conference on artificial intelligence, May 1990, Avignon France

[GAMT17] GAM T17-V2 : "Méthodologie de développement des logiciels intégrés dans les systèmes militaires"

[Morgue91] G. Morgue & T. Chehire : "Performances du couplage entre un ATMS et un générateur de systèmes experts", 11th International Workshop on expert systems and their applications, May 1991, Avignon France

[Schang86] T. Schang : "Threat identification : an Artificial Intelligence approach" ; 52nd Symposium AVP AGARD, September 1986, Firenze Italy

[Schang88a] T. Schang & F. Fages : "Real-time expert system for onboard radar identification" ; 55th Symposium AVP AGARD, April 1988, Cesme Turkey

[Schang88b] T. Schang : "Identification de radars par système expert", 8th International Workshop on expert systems and their applications, May 1988, Avignon France

[XIA91] Thomson-CSF : "XIA: Reference Manual" 1991

LE DEVELOPPEMENT DES LOGICIELS DE COMMANDES DE VOL, L'EXPERIENCE RAFALE

par

D. BEURRIER, F. VERGNOL et Ph. BOURDAIS.
Dassault Aviation - Direction Générale Technique
Division des Etudes Avancées - Département Dynamique du Vol
78 quai Marcel Dassault
92552 Saint-Cloud Cedex
France

RESUME

Après une présentation du système de commandes de vol (SCV) du RAFALE, cet exposé décrit les travaux menés par Dassault Aviation dans le domaine du **génie logiciel appliqué aux systèmes critiques du point de vue de la sécurité**, en mettant l'accent sur les points suivants :

- La méthodologie de développement : son originalité réside dans l'ajout d'une étape de formalisation des spécifications de logiciel, dont l'objectif est de renforcer la qualité du dialogue entre deux mondes distincts, les automaticiens d'une part, les informaticiens de l'autre.
- Les outils clés qui supportent cette méthodologie :
  - GISELE (Ref.1), outil de spécification, manipulant un langage formel, dont la puissance de test et la possibilité de prototypage automatique garantissent la qualité de la description,
  - VALIRAF, outil de validation, qui permet de comparer automatiquement le système réalisé aux modèles élaborés en phase d'étude et de spécification, et qui assure la gestion des essais dans le but d'en évaluer le taux de couverture.

Un bilan de l'expérience acquise et une présentation des perspectives de travaux futurs concluent cet exposé

INTRODUCTION

L'accroissement de complexité des fonctions intégrées dans les systèmes de commandes de vol a nécessité l'introduction de calculateurs numériques pour les mettre en oeuvre. Cet organe de calcul a posé un nouveau problème de sécurisation des systèmes. En effet, alors que les techniques de sécurisation des éléments matériels sont connues et éprouvées (redondance, vote ...), la sécurisation des logiciels reste, aujourd'hui encore, un sujet ouvert.

A l'instar des techniques de sécurisation des matériels, certains recommandent le recours à la réalisation de **logiciels N-Versions**, fonctionnellement équivalents, installés dans les N calculateurs du système, alors que d'autres préconisent la réalisation d'un **logiciel Zéro-faute**, répliqué à l'identique dans les calculateurs.

Dassault Aviation, bénéficiant de l'expérience acquise au cours du développement de plusieurs avions dotés de calculateurs numériques assurant des fonctions critiques, a opté pour l'approche logiciel Zéro-faute. Ce choix, conforté lors du développement du démonstrateur RAFALE A, suivi de l'avion de combat opérationnel RAFALE D, a mis en évidence la nécessité d'intégrer dans une même méthodologie les travaux de conception fonctionnelle (Ref.2) et de développement de logiciel.

Cette méthodologie, dont la clé réside dans l'étape de formalisation des spécifications de logiciel, renforce l'intégration des différentes équipes impliquées, depuis la conception jusqu'aux essais en vol. Elle a été élaborée dans un contexte particulier (Dassault Aviation développe entièrement les SCV de ses avions), mais s'étend aujourd'hui à des systèmes développés en coopération ou en sous-traitance.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

LE SYSTEME DE COMMANDES DE VOL DU RAFALE

Le Système de Commandes de Vol du RAFALE assure le contrôle de l'avion au moyen d'un ensemble de 11 gouvernes auquel il faut ajouter les 2 moteurs pour certaines fonctions particulières. Ces gouvernes sont utilisées conjointement pour réaliser un certain nombre de fonctions, parmi lesquelles on distingue:

- La fonction de **contrôle du comportement non piloté**, qui assure les stabilisations artificielles statiques et dynamiques de l'avion, tant en longitudinal qu'en transversal. Cette fonction réalise également le contrôle dynamique des couplages entre le SCV et les modes avion (modes structuraux, modes de l'avion sur ses atterrisseurs).

- La fonction de **contrôle de comportement piloté**, qui optimise la réponse de l'avion aux ordres de pilotage quelle que soit la phase du vol (croisière, combat, approche, ravitaillement, ...) tout en minimisant la charge de travail du pilote liée au respect des limites aérodynamiques (incidence, dérapage) et structurales (facteur de charge, ...) de l'avion, la protection vis à vis de ces limites étant réalisée de façon automatique.

- La fonction de **contrôle de configuration**, qui optimise l'utilisation de l'ensemble des gouvernes pour adapter au mieux la configuration aérodynamique et structurale de l'avion aux conditions de vol et aux ordres de pilotage de façon à exploiter au mieux ses capacités de manoeuvrabilité (marge et limite de manoeuvre, vitesses de roulis, ...) et ses performances (minimisation de la traînée, vitesses d'approche, ...).

- La fonction de contrôle de l'avion marin au catapultage.

- Le couplage du contrôle de l'avion à certaines fonctions opérationnelles (vol à très basse altitude, approche automatique).

Une part importante de la complexité des traitements fonctionnels réalisés par le SCV découle de la nécessité de réaliser la synthèse des ordres élaborés par ces diverses fonctions tout en conservant, en cas de conflit ou en limite d'autorité du système, la priorité aux fonctions essentielles (stabilisations, limitations automatiques, ...).

L'intégration de l'ensemble de ces fonctions au sein d'un système de Commandes de Vol Electriques permet donc d'optimiser le comportement général de l'avion, avec pour conséquence l'homogénéisation de ce comportement vu du pilote, et ce pour l'ensemble étendu des configurations utilisées (masse, centrage, emports, ...) et dans tout le domaine de vol.

Cette intégration représente un volume de traitements que seul un système numérique doté de calculateurs de forte puissance est à même de fournir. Ces calculateurs exploitent les informations délivrées par un ensemble de capteurs, mesurant les ordres du pilote et les mouvements de l'avion, pour élaborer les consignes à destination des actionneurs qui pilotent les gouvernes.

Se basant sur l'expérience acquise par Dassault Aviation dans le domaine des Commandes de Vol entièrement électriques avec les programmes M2000, M4000 et MIII NG, expérience approfondie, notamment dans le domaine de l'utilisation de calculateurs numériques, par l'expérimentation du démonstrateur RAFALE A, le Système de Commandes de Vol du RAFALE est doté d'une architecture, compatible de son haut niveau de criticité, basée sur une redondance d'ordre 4 de ses constituants principaux. Cette redondance est exploitée de manière optimale par le biais de reconfigurations automatiques en fonction de l'état d'intégrité des différents constituants du Système.

Dans cette architecture, on distingue : (Planche 1)

- Les capteurs, de redondance 4 pour les plus critiques (capteurs pilotes, gyromètres, accéléromètres, ...), de redondance 3 pour les autres (capteurs anémo-baro-clinométriques, ...).

- Les traitements fonctionnels, de redondance 4, constitués de 3 chaînes principales numériques, d'architecture matérielle et logicielle identique, associées à une chaîne de secours analogique indépendante.

- Les actionneurs, constitués pour les gouvernes principales (élevons et drapeau) de servocommandes électro-hydrauliques double corps à 4 entrées.

L'ensemble de ces éléments est associé à 4 alimentations électriques et 2 circuits hydrauliques.

LA METHODOLOGIE

La réalisation de nouvelles fonctions de contrôle du vol a été rendue nécessaire pour répondre à l'évolution des besoins opérationnels des avions d'armes modernes. La possibilité de contrôler des avions instables, la complexité croissante de ces fonctions, ont renforcé leur aspect critique du point de vue de la sécurité et imposé une mise à jour progressive de leur méthodologie de développement.

Les puissances de calcul mises en jeu, non réalisables par des technologies analogiques, ont imposé l'introduction de calculateurs numériques et le recours à une méthodologie de développement de logiciels qui apporte la conviction, sinon la preuve, d'obtenir un logiciel sans défaut. Cette nécessité n'est pas liée au logiciel lui même mais à la fonction qu'il met en oeuvre, car si la fonction est critique du point de vue de la sécurité alors le logiciel l'est aussi.

Ces méthodologies, chacune dans leur domaine, apportent la garantie d'un développement correctement maîtrisé et minimisent le risque d'introduction d'erreurs. Force est de constater, et ceci bien qu'évident est souvent négligé tant par les habitudes de travail que par les normes actuelles, que la source la plus importante d'introduction d'erreurs se situe à la frontière de deux métiers.

En effet, alors que la conception fonctionnelle requiert des compétences en dynamique du vol et en automatique, le développement de logiciel nécessite des compétences en informatique. Nous sommes alors face à un problème de communication entre deux mondes, qui s'ils ne s'ignorent pas, bien souvent ne se comprennent pas.

Partant de cette analyse, Dassault Aviation a conçu une méthodologie qui fédère l'ensemble du cycle de développement, depuis l'étape initiale d'étude fonctionnelle jusqu'à la validation finale du système réalisé. Tout en apportant une solution au problème de communication, le caractère de **continuité du cycle de développement** entraîne des possibilités de vérification accrues par la réutilisation dans une phase quelconque des jeux de tests réalisés dans les phases précédentes.

La méthodologie comporte quatre grandes étapes (Planche 2) :

- CONCEPTION : la Définition Fonctionnelle (description des lois de contrôle) constitue la sortie principale de cette étape. Les études de définition s'appuient sur le développement d'un Modèle Fonctionnel (forme exécutable des algorithmes, **MF1**). L'activation de ce modèle, dans un contexte de simulation aussi réaliste que possible, constitue la Vérification Fonctionnelle (**VF**) des algorithmes au regard des Spécifications Globales, et clôt l'étape de conception. Il est important de noter que, dès cette étape, l'utilisation d'un simulateur temps réel permet de faire participer les pilotes à la validation des choix de conception.

- FORMALISATION : c'est l'étape de transition entre le monde des automaticiens et celui des informaticiens. C'est pourquoi le langage formel utilisé a été conçu pour être manipulé et compris par ces deux mondes. Cette étape est fondamentale car son but est multiple :

  - c'est une **revue de définition fonctionnelle** : pour cette raison elle est confiée à une équipe ayant les mêmes compétences que l'équipe de conception. Ce point est essentiel car il permet d'assurer une relecture active de la conception dans un but de reformulation.

  - elle produit une **Définition Technique Fonctionnelle Formelle** (DTFF) qui est la traduction formelle du contenu de la Définition Fonctionnelle, dont les qualités (complétude, cohérence, ...) peuvent être vérifiées automatiquement par l'outil GISELE.

- un **modèle exécutable** (MF2) de la DTFF est obtenu automatiquement et est utilisé pour :
  - vérifier la conformité de la DTFF vis à vis de la Définition Fonctionnelle. Cette vérification est réalisée par comparaison du comportement de MF1 et MF2 (VL)
  - définir les tests unitaires et d'intégration fonctionnellement représentatifs.

La conformité de la DTFF étant vérifiée, l'étape de réalisation est engagée.

- REALISATION : le développement et l'implantation du logiciel dans l'équipement sont simplifiés. On tire dans cette étape un grand bénéfice de la formalisation car :

  - une grande partie du logiciel est obtenue par **codage automatique** de la DTFF, avec pour conséquence un impact favorable sur le coût et la sûreté de l'opération,

  - l'intégration du logiciel dans l'équipement est facilitée par le jeu de tests élaboré dans l'étape de formalisation. Cette tâche constitue une **prévalidation** du logiciel (VI).

La prévalidation du logiciel effectuée, on quitte l'étape de réalisation pour entrer dans celle de validation.

- VALIDATION : l'équipement réel est installé dans un environnement de simulation fonctionnellement représentatif qui comprend des équipements réels (capteurs, servocommandes, ...) et une modélisation temps réel de l'environnement (avion, vent, turbulence, autres systèmes, ...). Au cours de cette étape, des essais sont réalisés par des pilotes d'origines différentes (ingénieurs, pilotes d'essais, ...). Il s'agit donc d'une validation globale du système qui présente deux aspects :

  - **vérification fonctionnelle** (VF) vis à vis des Spécifications Globales, grâce à une campagne d'essais définie lors de l'étape de conception. Les essais réalisés sont suivis en temps réel par les ingénieurs concepteurs. Simultanément, les signaux nécessaires (entrés, sorties, états internes de l'équipement) sont systématiquement enregistrés. Ils sont réutilisés en temps différé pour être injectés dans un modèle de comportement du système complet, de manière à s'assurer que les performances obtenues sont conformes à la Spécification Globale (Planche 3)

  - **vérification du comportement du logiciel** (VL), par une comparaison avec les modèles MF1 et MF2 soumis aux mêmes sollicitations. La majeure partie de cette activité est réalisée automatiquement par l'outil VALIRAF, que nous présentons en détail dans la suite de cet exposé.

C'est à la vue des résultats de validation que l'on autorisera la mise en vol de l'avion.

L'OUTIL GISELE

L'outil GISELE (Génération Interactive de Spécifications d'Ensemble Logiciel Embarqué) a été développé pour assister les ingénieurs dans la rédaction de la DTFF, en leur apportant des facilités de manipulation d'un langage formel que nous avons conçu en tenant compte des contraintes suivantes :

- le langage doit permettre de décrire les traitements à implanter en restant indépendant de la machine cible (matériel et langage de programmation),

- le langage doit être compréhensible par des personnes qui ne sont pas des professionnels de l'informatique,

- le langage doit permettre de réaliser des spécifications détaillées dépourvues d'ambiguïté,

- les spécifications ainsi réalisées doivent être "facilement" validables.

Pour répondre à ces contraintes, nous avons élaboré une modélisation de l'objet à spécifier. Ce modèle, tout à fait général, a permis de définir les principaux éléments du langage qui, par la suite, ont été affinés pour aboutir à la création des entités suivantes :

- SYSTEME (Planche 4) : son architecture met en jeu une ou plusieurs **ressources** communiquant entre elles par un **réseau**. Le "reste du monde" est considéré comme une ressource qui n'a pas à être spécifiée,

- RESEAU : constitué de "**boites à lettres**", il gère les informations échangées par les ressources lors d'actions élémentaires d'écriture ou de lecture de **messages**. La chronologie des échanges peut être contrainte par le temps (datation),

- RESSOURCE : elle correspond à un traitement numérique individualisé, indépendant des traitements des autres ressources. Une ressource est une entité assurant un traitement séquentiel dont la description peut être hiérarchisée en **fonctions** (Planche 5),

- FONCTION : elle est décrite selon une approche descendante par décomposition en **modules** organisés selon une arborescence strictement hiérarchique,

  - la fonction de plus haut niveau, appelée fonction principale, constitue le point d'entrée dans la description des traitements d'une ressource. C'est la seule autorisée à échanger des informations via le réseau,

  - les autres fonctions, appelées fonctions secondaires, sont munies d'une interface strictement définie permettant d'assurer que leur utilisation est cohérente vis à vis de leur définition,

- MODULE (Planche 6) : c'est l'élément de base de la description des traitements. Réalisé sous forme d'un **organigramme**, le module comporte en outre des informations générales (résumé fonctionnel en langage courant, date de mise à jour ...) et la liste des **identificateurs** qu'il manipule,

- ORGANIGRAMME : constitué d'un ensemble de cadres (qui renferment les **expressions**) et de liaisons (qui figurent la logique de séquencement), il est soumis à des règles de syntaxe adaptées aux exigences de sécurité dont les principales sont :

  - entièrement contenu sur une page A4,
  - un chemin unique d'entrée, un chemin unique de sortie,
  - trois structures de graphe (Planche 7),

- EXPRESSION : elle décrit les opérations à effectuer sur les informations. Les opérateurs autorisés sont limités à :

  - affectation (:=),
  - arithmétique (+, -, *, /),
  - logique (NON, ET, OU, EQV),
  - relationnel (=, /=, >, >=, <, <=),

- IDENTIFICATEUR : il associe un nom (8 caractères maximum) à une information. Chaque nom est accompagné de **déclarations** parmi lesquelles on trouve :

  - un commentaire,
  - un type (variable, état, donnée ...),
  - un format (réel, entier ...),
  - une unité physique,
  - les valeurs de dimensions (les tables sont limitées à quatre dimensions),
  - les valeurs (les bornes de variation pour les informations dynamiques).

La structure du langage, répond aux propriétés suivantes :

- GENERALITE : la possibilité de décrire des traitements parallèles ou séquentiels avec ou sans contraintes temps réel est offerte,

- FORMALISME : par sa rigueur, il permet de contrôler la qualité de la description, et garantit contre la possibilité d'aboutir à un logiciel dont le comportement ne serait pas sûr (écrasement, blocage ...). De manière à faciliter l'établissement d'une DTFF, des conventions du langage permettent l'introduction de description en langage usuel (**traitement commenté**). Bien entendu, ces traitements ne sont pas testables, mais leur intégration dans une description formelle ne supprime en aucun cas les possibilités de test de cette dernière,

- LISIBILITE : basée sur une description graphique (organigrammes), la partie textuelle manipule des expressions mathématiques usuelles. Les détails (déclarations, valeurs ...) sont déportés dans des descriptions en annexe,

- TESTABILITE : il ouvre des possibilités de test importantes parmi lesquelles on trouve :

  - test structurel (hiérarchie, graphe des chemins),
  - test syntaxique (graphique, textuel),
  - test de cohérence (interface de fonction, unité physique),
  - test de flot de données (antécédence, débordement de table),
  - test de complétude.

- CAPACITE DE PROTOTYPAGE : l'obtention automatique d'un prototype exécutable permet d'assurer que la spécification du logiciel à produire est conforme au besoin.

Réalisé en Fortran, excepté la base de données qui est en assembleur, l'outil GISELE est implanté sur les ordinateurs de nos centres de calcul (IBM ES/9000).
L'ergonomie du poste de travail (écran IBM 5080) repose sur des techniques conventionnelles (multi-fenêtrage, menus contextuels ...) et sur un éditeur syntaxique d'organigramme (graphique avec placement automatique des cadres et des liaisons) (Planche 8).
La production automatique de document est faite sur traceurs électrostatiques (Planche 9).

La totalité des règles imposées par le langage est vérifiée par des fonctions de **tests statiques** dont certaines sont peu ou pas implantées dans les outils du marché.
On cite ici pour exemple :

- **le test d'antécédence** : qui consiste à vérifier que tous les chemins de calcul aboutissant à l'utilisation d'une information ont vu cette information définie. Le dual consistant à vérifier que toute information définie est suivie d'une utilisation sur au moins un chemin,

- **le test d'homogénéité** : qui grâce à l'unité physique fournie dans les déclarations d'identificateur assure que toutes les expressions sont physiquement homogènes.

Cependant, le respect des règles du langage n'est pas suffisant pour vérifier que ce que l'on a décrit correspond bien à ce que l'on veut développer. Grâce à la fonction de **génération automatique de code** exécutable, cette vérification est possible.
Le code généré peut être **instrumenté automatiquement** (insertion de tests, compteurs d'activations de chemins). L'instrumentation ainsi réalisée est utilisée pour s'assurer de l'exhaustivité des tests d'intégration et, reprise par VALIRAF, permet de mesurer le taux de couverture des essais de validation.
Enfin, la production d'un **code intermédiaire** ("pseudo-code") permet, moyennant le développement du traducteur approprié, la réalisation automatique du logiciel cible.
Les logiciels de commande de vol des avions de type RAFALE D ont été développés par cette procédure.

L'OUTIL VALIRAF

L'outil VALIRAF a été développé afin de réaliser d'une façon quasi-automatique la tâche de validation logicielle du logiciel du système de commandes de vol.
On rappelle que cette tâche s'inscrit dans l'étape de VALIDATION où l'équipement réel est installé et essayé au banc global de simulation. A titre indicatif le volume d'essais réalisés et exploités représente plusieurs centaines d'heures.

L'outil VALIRAF est constitué de plusieurs fonctionnalités dont la procédure de mise en oeuvre est précisée planche 10.

Il répond à deux missions essentielles :

- **La validation logicielle** proprement dite, fondée sur la comparaison du logiciel embarqué avec les modèles MF1 et MF2 :

  Lors des essais au banc global de simulation, l'ensemble des signaux nécessaires à la validation logicielle, tels que toutes les entrées, toutes les sorties, tous les états internes plus quelques variables intermédiaires sont enregistrés sur bande magnétique. Ces signaux subissent un pré-traitement, destiné essentiellement à vérifier l'intégrité des mesures, puis sont utilisés pour solliciter les modèles MF1 et MF2.

Les différences des valeurs élaborées, à chaque cycle de calcul, par le logiciel et les deux modèles MF1 et MF2 pour un même terme fonctionnel sont comparées à des seuils prédéfinis. En cas de dépassement, les écarts correspondants sont signalés et mémorisés.

Tout écart signalé au cours du traitement global fait l'objet d'une analyse détaillée. Cette analyse est, dans certains cas simples et bien identifiés, réalisée automatiquement par l'outil VALIRAF. Dans le cas contraire, elle est réalisée par l'ingénieur au cours d'un traitement partiel des essais au voisinage des instants signalant les écarts. Le traitement consiste alors à décomposer le terme signalé en ses différents constituants pour chacun des modèles MF1 et MF2, à repérer sans ambiguïté le terme amont composant l'origine de l'écart, puis finalement à élaborer un diagnostic. Durant une campagne de validation logicielle, tous les écarts doivent être analysés et expliqués afin d'assurer une vérification complète et totale de la Définition Technique Fonctionnelle Formelle ainsi que de sa réalisation.

- <u>La gestion dynamique des essais de validation</u> destinée à assurer et mesurer la complétude des essais réalisés :

  Au cours du traitement global défini ci-dessus, sont archivés en Bases de Données (BDD) certains paramètres représentatifs du vol d'une part (point de vol, mode de fonctionnement du SCV...) et de l'activité ou la non-activité des branches fonctionnelles d'autre part.

  Cette dernière information est obtenue par exploitation des compteurs d'activation de chemins du module MF2 dont l'outil GISELE a assuré l'instrumentation automatique.

  L'exploitation systématique de la BDD permet ,en cours de validation globale, de compléter et d'orienter les essais vers des programmes susceptibles de parcourir les branches fonctionnelles non-activées précédemment et/ou d'observer le fonctionnement du système avion + SCV dans un ensemble de conditions de vol représentatives de son utilisation réelle.

  En fin de campagne de validation, l'exploitation de la BDD fournit une mesure du taux de couverture des essais réalisés.

Réalisé exclusivement en Fortran, l'outil VALIRAF est aussi implanté sur les ordinateurs de nos centres de calcul (IBM ES/9000). La BDD est gérée par le Système de Gestion de Base de Données Relationnel ORACLE.

VALIRAF est constitué d'un noyau qui regroupe tout un ensemble de sous-programmes Fortran permettant de réaliser d'une façon automatique les actions de lecture et de vérification de validité des blocs de données, la comparaison des paramètres surveillés avec les seuils prédéfinis ainsi que l'élaboration des diagnostics automatiques, l'initialisation des modèles MF1 et MF2, le tracé des courbes, l'élaboration de statistiques associées à la surveillance des écarts, à quoi s'ajoutent les sous-programmes propres à la BDD tels que l'initialisation, le stockage des paramètres de vol et le stockage des indicateurs de branches de MF2. Ce noyau est invariant vis à vis de toute évolution de logiciel puisqu'il réalise des tâches propres à la procédure de validation logicielle et indépendantes de la nature du logiciel.

Un environnement conversationnel interactif invariant vis à vis du logiciel permet d'activer le traitement global systématique des vols et les stockages en BDD, puis de mener à bien le traitement partiel d'une portion de vol si le traitement global a mis en évidence des écarts, ainsi que l'interrogation de la BDD.

Un certain nombre d'éléments adaptés au logiciel à valider, comme les modèles MF1 et MF2 ainsi que des fichiers d'interfaces enregistrements / modèles MF1 et MF2, font partie intégrante de l'outil.

L'EXPÉRIENCE ACQUISE

Début 1976, le développement du "MYSTERE 20 REGLEMENTATION" (simulateur volant d'avion de transport) avait donné lieu à l'ébauche d'une méthode de développement de logiciels embarqués critiques.
Le "MIRAGE 2000 NUM" (développement exploratoire) a été à l'origine de la version 1 de GISELE dont la première utilisation a eu lieu en Octobre 1983.

Riche de cette expérience, la version 2 de GISELE a été développée et mise en service en Janvier 1985 pour le RAFALE A, accompagnée en Avril 1986 de la première versicn de VALIRAF.
Les retours des utilisateurs ont permis d'améliorer ces outils et de les livrer en Décembre 1988 (GISELE 3) et en Septembre 1990 (VALIRAF 2) dans le cadre du programme RAFALE D dont les événements marquants sont les suivants :

- premier vol du C01 en Mai 1991
- premier vol du M01 en Décembre 1991,
- premier vol du B01 fin Avril 1993,
- première campagne porte-avions du M01 en Avril-Mai 1993.

Le déroulement de ce programme démontre la capacité de Dassault Aviation à satisfaire des objectifs techniques complexes dans le respect des délais annoncés et confirme sa parfaite maîtrise dans le domaine du développement des logiciels embarqués critiques.

LES PERSPECTIVES

Les méthodes et les outils du génie logiciel, dans toutes les étapes du cycle de vie des produits développés, connaissent actuellement un développement spectaculaire.

En ce qui concerne l'étape de spécification de logiciel, la disponibilité d'une spécification formelle, telle que celle qui est produite à l'aide de l'outil GISELE, constitue la base essentielle des travaux de génie logiciel à venir.
Ces travaux seront conduits dans les perspectives suivantes :

- Accroissement des possibilités de test de la spécification produite.
- Mise en oeuvre d'outils de preuve. Ces derniers, aujourd'hui de possibilités restreintes et d'emploi difficile, pourront, dans les années à venir être appliqués à des traitements de plus en plus étendus.

Par ailleurs, quel que soit le niveau de qualité de la spécification, et le niveau de qualité du logiciel embarqué produit (dont la réalisation sera de plus en plus automatisée), le besoin de validation du système réalisé subsistera.

Dans ces conditions, l'évolution de l'outil VALIRAF sera conduite selon les deux perspectives principales suivantes :

- Automatisation de la préparation et de la réalisation des campagnes de validation.
- Automatisation de la gestion et de la synthèse des informations acquises au cours de ces campagnes.

Enfin, un effort constant continuera à être dédié à l'intégration des méthodes et moyens présentés ici (adaptés particulièrement aux applications critiques du point de vue de la sécurité) avec les environnements de développements les plus répandus, afin que :

- Par l'emploi d'Ateliers de Génie Logiciel bien adaptés et efficaces, les coûts de développements des applications soient aussi réduits que possible.
- Les actions de coopérations dans le domaine du développement de systèmes embarqués puissent être engagées dans les meilleures conditions.

CONCLUSION

Dans l'approche Dassault Aviation du développement de systèmes critiques du point de vue de la sécurité, les deux points délicats que sont le transfert du besoin fonctionnel aux réalisateurs et la mesure de la complétude des essais de validation sont couverts.

Nous ne prétendrons pas que la méthodologie de développement et les outils que nous avons présentés constituent l'unique solution pour obtenir la qualité escomptée, mais le déroulement du programme RAFALE en démontre l'efficacité.

REFERENCES

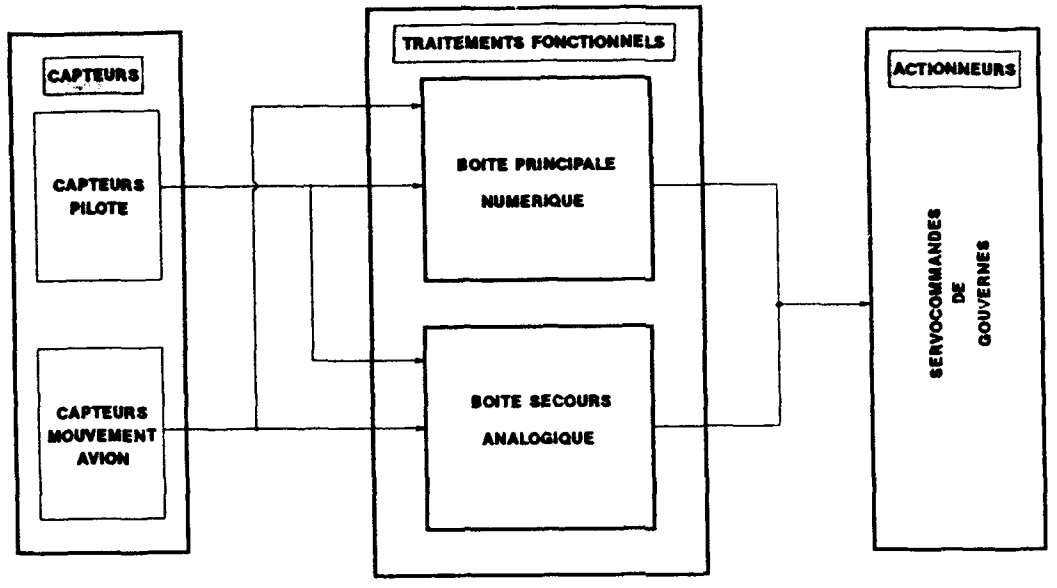
1 GISELE (Génération Interactive de Spécifications d'Ensemble Logiciel Embarqué)
J. CHOPLIN et D. BEURRIER (Dassault Aviation) AGARD-FMP , Octobre 1984, Toronto (Canada).

2 Méthode de développement du système de contrôle du vol du RAFALE.
Ph. BOURDAIS et R-L. DURAND (Dassault Aviation) AGARD-FMP , Mai 1992, Chania, Crète (Grèce).

DEVELOPPEMENT DES LOGICIELS CRITIQUES

ARCHITECTURE MATERIELLE DU SCV DU RAFALE

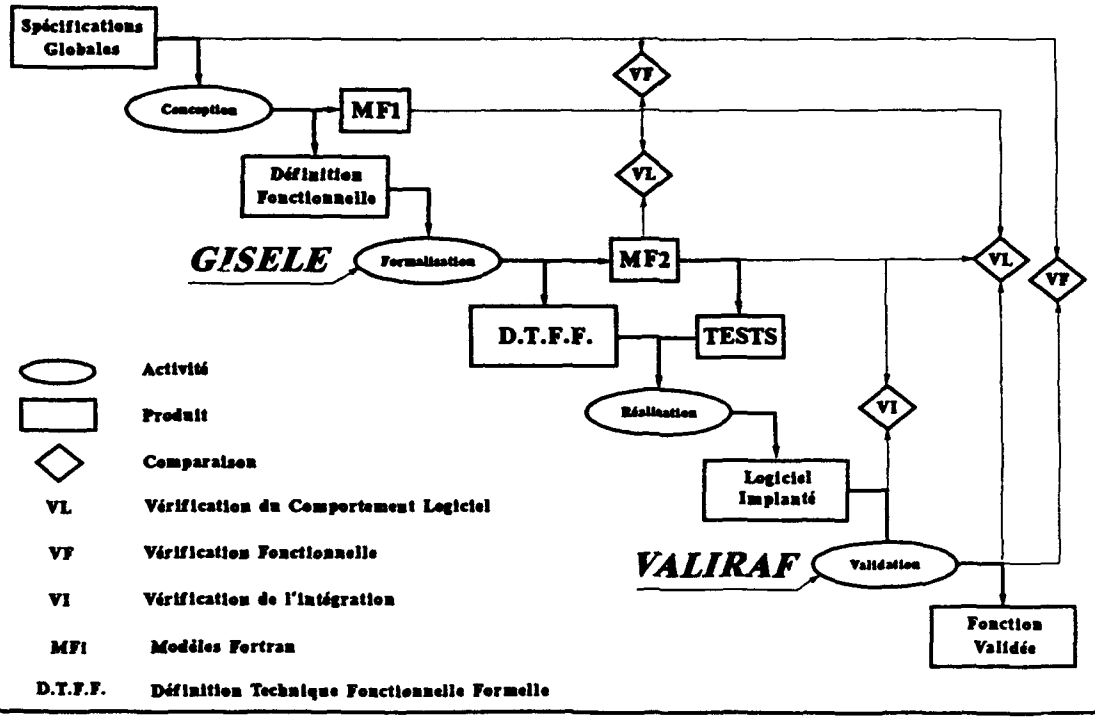
AGARD / AVP | MAI 93 | PL : 01



DEVELOPPEMENT DES LOGICIELS CRITIQUES

METHODE

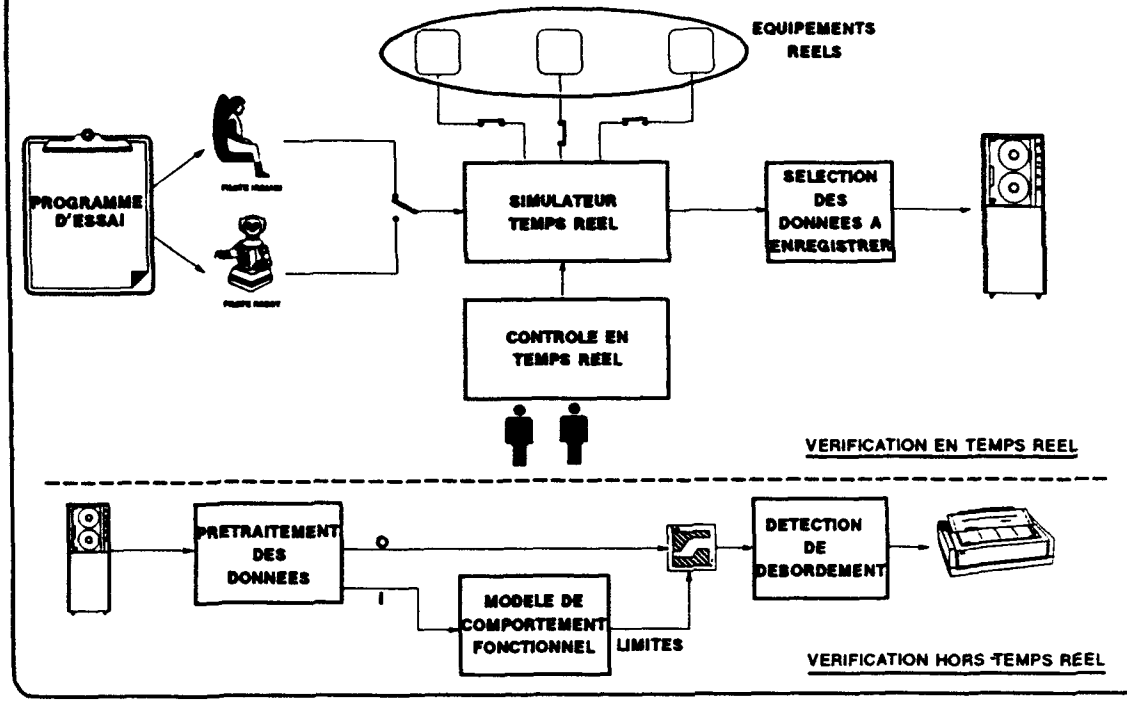
AGARD / AVP | MAI 93 | PL : 02



DEVELOPPEMENT DES LOGICIELS CRITIQUES

VERIFICATION FONCTIONNELLE

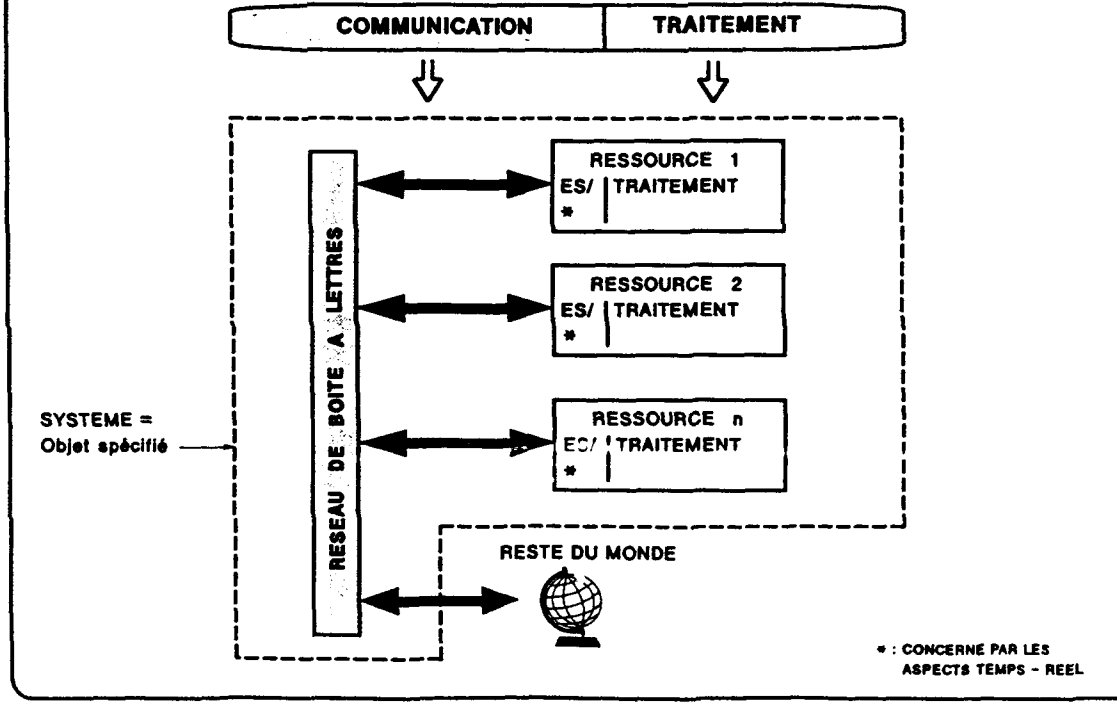
AGARD / AVP | MAI 83 | PL : 03



DEVELOPPEMENT DES LOGICIELS CRITIQUES

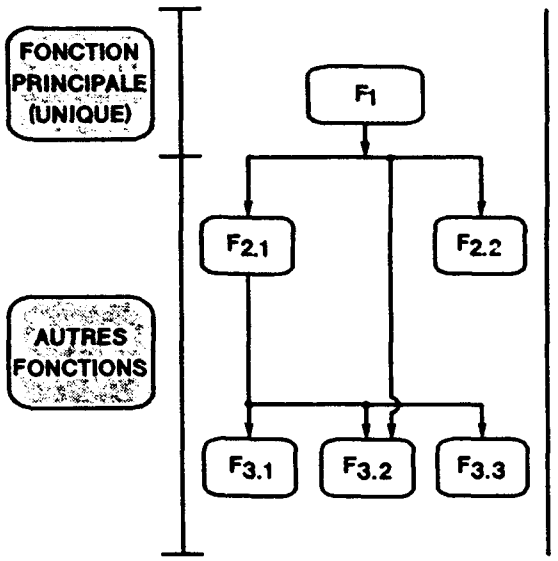
LE SYSTEME

AGARD / AVP | MAI 83 | PL : 04

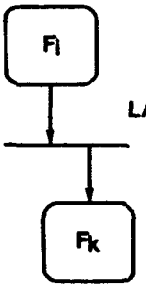


DEVELOPPEMENT DES LOGICIELS CRITIQUES

HIERARCHIE DES FONCTIONS



SYMBOLOGIE

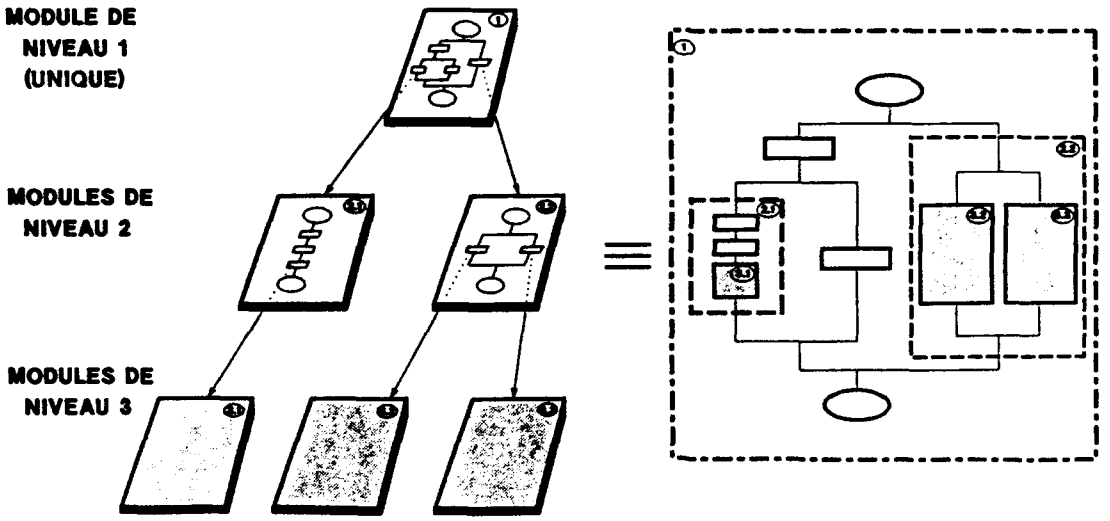


LA FONCTION Fk :

- RECOIT LE FLOT D'ENTREE DE LA FONCTION Fi
- ELABORE LE FLOT DE SORTIE
- RETOURNE LE FLOT DE SORTIE A LA FONCTION Fi

DEVELOPPEMENT DES LOGICIELS CRITIQUES

HIERARCHIE DES MODULES



DEVELOPPEMENT DES LOGICIELS CRITIQUES

ELEMENTS DE SYNTAXE GRAPHIQUE

AGARD / AVP | MAI 93 | PL : 07

SEQUENCES

TRAITEMENT DECRIT EN LANGAGE NATUREL

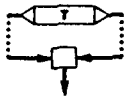
TRAITEMENT DECRIT EN LANGAGE FORMEL

APPEL DE FONCTION

REFERENCE A UN MODULE D'ENTREE/SORTIE

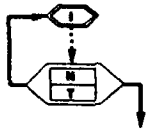
REFERENCE A UN MODULE DE TRAITEMENT

TESTS



DESCRIPTION FORMELLE DE L'EXPRESSION TESTEE

BOUCLES



I : NOM DU COMPTEUR DE BOUCLE

N : NOMBRE MAXIMUM D'ITERATIONS (DONNEE)

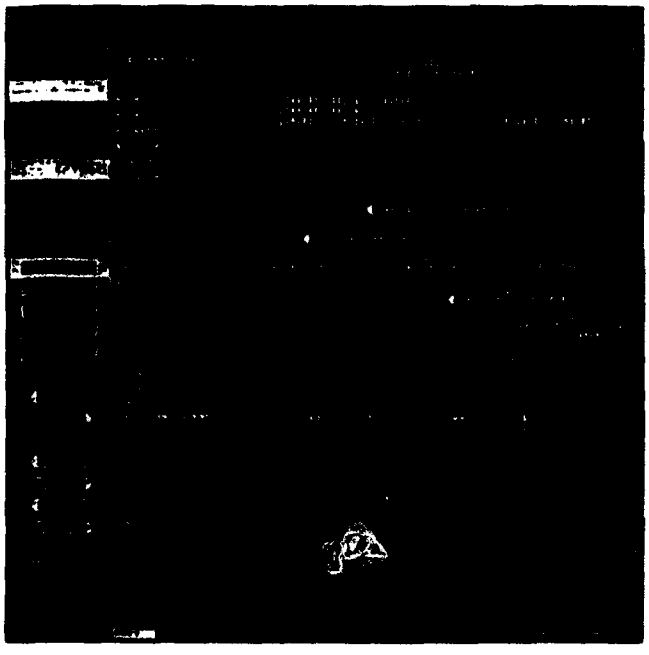
T : TEST DE SORTIE ANTICIPEE

I ET T SONT OPTIONNELS

DEVELOPPEMENT DES LOGICIELS CRITIQUES

EXEMPLE DE VISUALISATION

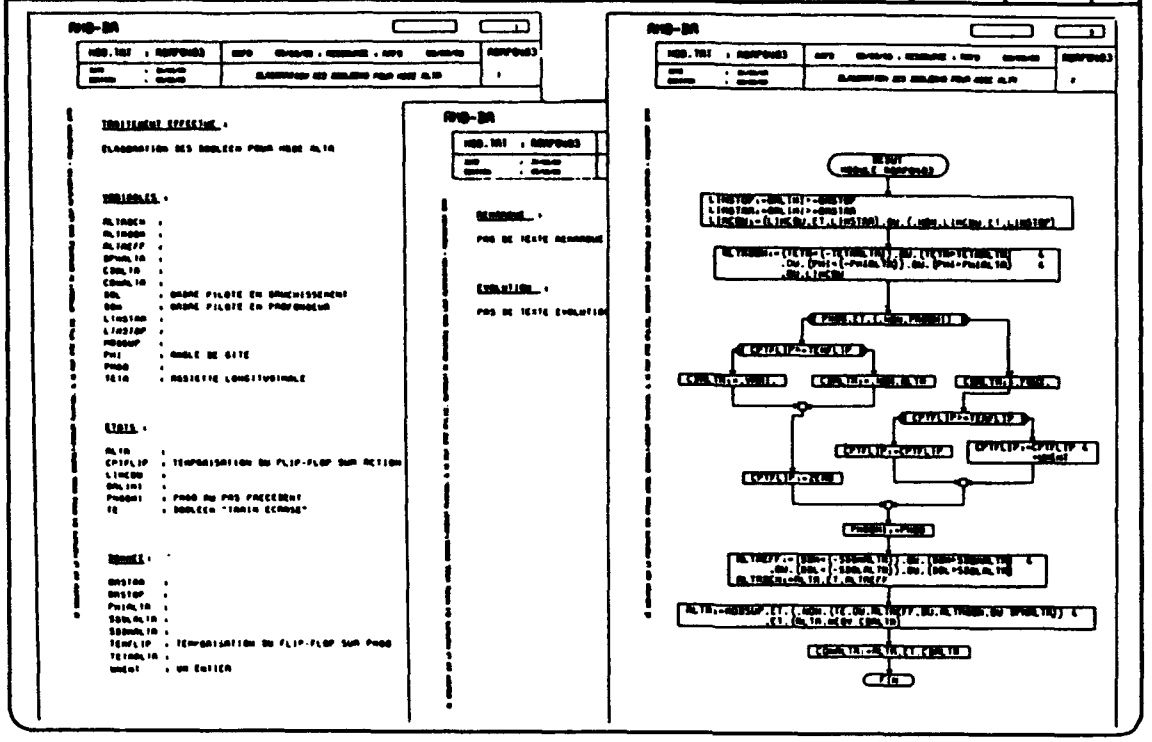
AGARD / AVP | MAI 93 | PL : 08



DEVELOPPEMENT DES LOGICIELS CRITIQUES

EXEMPLE D'EDITION

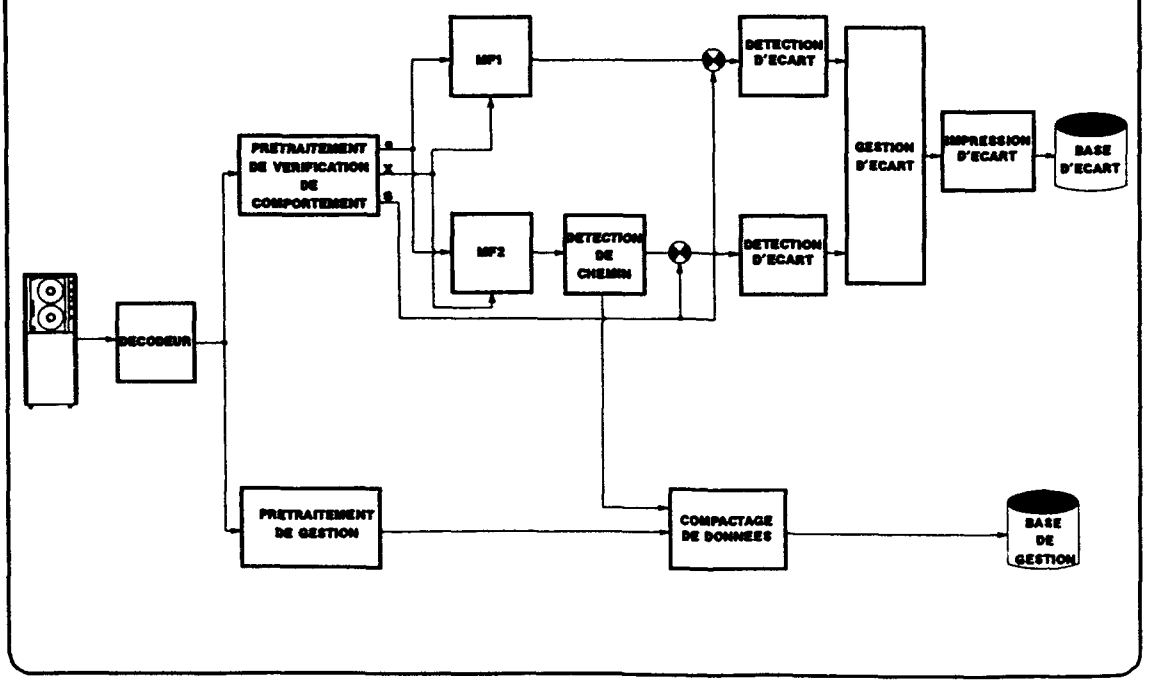
AGARD / AVP | MAI 93 | PL : 09



DEVELOPPEMENT DES LOGICIELS CRITIQUES

VERIFICATION DE COMPORTEMENT DU LOGICIEL

AGARD / AVP | MAI 93 | PL : 10



OBJECT VERSUS FUNCTIONAL ORIENTED DESIGN

P. Occelli
Alenia S.P.A.
Corso Marche, 41
10146 TURIN (ITALY)

1 INTRODUCTION

Since the early Eighties the Object Oriented approach to software system development was proposed as a possible solution of the so called 'software crisis'.
The claimed benefits were that Object Oriented Design (OOD) had the potential to improve software quality by making possible a direct and natural correspondence between the real world and its model.

Many variants of the original approach were proposed and the new trend was spread across the Software Engineering community.
The 'traditional' functional oriented methods were suddenly considered out of date and not appropriate to support development of large real time systems.
There were, of course, some drawbacks but they were always attributed to method immaturity and poor tool support, two self-solving problems as time passed.
Ten years or more have gone since then and OOD methods have been widely adopted for the development of large distributed real time systems.

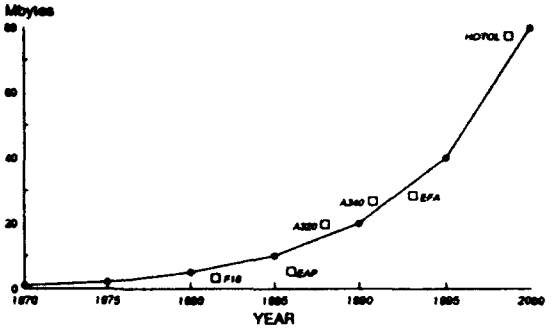
Are the lessons learnt from such projects according to the expectations?
Are Functional Oriented methods still able to support software projects of the size required by the Aerospace industry in the years leading to the 2000?

This paper proposes a possible answer to these questions comparing the pro and cons of both methods.
This comparison will be carried out on issues like transition from requirements to design and to Ada code, traceability from requirements to design, software safety, software maintainability, software testing and relationship with DOD-STD-2167.
Let's start with some general principles and with a brief summary of the salient characteristics of both methods.

2 THE SOFTWARE CRISIS

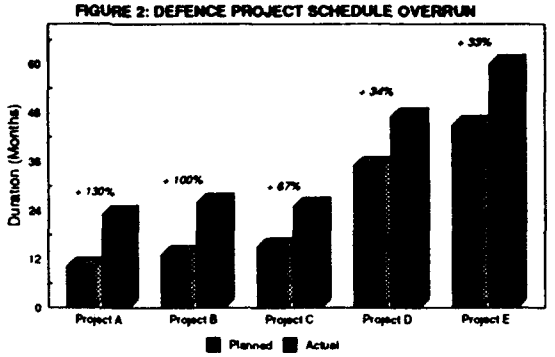
The increasing complexity of onboard navigation, guidance and control systems, together with the trend to implement in software functions traditionally accomplished by hardware devices, has led to a situation where the size and complexity of the embedded software has become unmanageable.
Figure 1 shows the oustanding growth of on-board software for modern civil and military programs.

FIGURE 1: GROWTH OF ON-BOARD SOFTWARE



Indeed in the last years virtually all software projects run behind schedule, exceed their estimated costs and do not fully meet customer requirements.
This situation, known by the software community as the 'software crisis', results in software not meeting its requirements, being unreliable, too costly, difficult to change and maintain.
Figure 2 displays some examples of Defence projects that overrun their planned schedules.

FIGURE 2: DEFENCE PROJECT SCHEDULE OVERRUN



As each author tends to be innovative and doesn't want to be involved in copyright quarrels, the literature is full of different definitions for the 'software crisis'. However, the common factor to all definitions is the difficulty to manage the extreme complexity of the software embedded in such systems.

The design and implementation of systems consisting of millions of lines of code require an effort that is clearly beyond the intellectual and physical capacity of a single person. Of course, adding more people to a project increases the overhead due to communication and coordination problems. Furthermore, the fact that few people can understand the complete structure often makes the process to

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

modify such systems a nightmare. To exacerbate this obstacle the elusive nature of the software itself typically makes challenging even to focus and isolate the sensed problems.

Besides complexity, other acknowledged causes of the software crisis are the shortage of trained personnel and the tending of people to resist to any new trend and to continue using archaic methods and tools.
Indeed the use of software tools and techniques to compensate for the human limitation in managing software complexity is the key point for combating the crisis.
Proper and efficient use of such methods and tools is the discipline commonly known as Software Engineering.

3 PRINCIPLES OF SOFTWARE ENGINEERING

The first goal of any software/system, development is that the product meet the specified requirements.
Unluckily consistent and clear requirements are rarely available. Furthermore, virtually all software projects have to face strong constraints related to timescale, hardware development obstacles (e.g. target devices available late and of poor quality) and integration problems.

Among the outstanding features of the onboard real time software systems we can certainly indicate modifiability, efficiency (in term of time and space) reliability (for safety critical systems) and understandability.
To achieve these goals a set of software engineering principles should be applied, among others:

- Abstraction and Information Hiding
- Modularity and Localisation
- Completeness
- Testability

The complexity of a typical real time system is such that a leading factor for a successful project is an appropriate decomposition of the system into simpler and smaller modules.
To define consistent criteria for the representation of a real system and to support its decomposition countless methodologies, more or less supported by tools, have been developed.
Virtually all methods can be divided in two broad categories, namely the more traditional Functional methods, also known as process-oriented or structured design and the Object Oriented methods.
The latest trend of some segment of the Software Engineering community is to consider the latter the only effective answer to the software crisis. Indeed OOD supporters tend to split the universe in OOD and non-OOD methods.
To represent the structure of the system under development Functional methods adopt a decomposition resulting in design elements being composed by a bunch of processes while in an OOD the design elements directly map real world entities.

In order to better understand the scope and content of the following paragraphs I would like to spend a few words on one of the most important features of a real time system: **CONCURRENCY**.

Concurrency can be defined as the capability to run more than one thread of execution at the same time on a single CPU, thus implementing a virtual parallelism.
Let's make an example.
An aircraft utility system may be tasked to control the speed, the oil temperature and oil pressure of the engine, the turbine and the related gearbox. Assuming the system encompasses a single processor, that results in a total of six different activities (functions) to be performed continuously on the same CPU. This means that the functions shall run in their time slices allocated according to system requirements and scheduling strategy. In our example the six functions could be grouped in three processes, controlling respectively temperatures, pressures and speeds. The iteration rate of each process will depend on the features of its implemented functions; e.g. the temperature control will have less stringent timing requirements than the turbine speed control, where a delay of few milliseconds can result in over-speed and consequent hardware damages. Typically, processes controlling temperature and oil pressure run at 5 to 10 Hz, while turbine speed control may require up to 200 Hz.
To design and implement a scheduling mechanism to manage concurrent issues, possibly with the use of hardware related resource (e.g. interrupts) and/or programming language features (e.g. Ada tasking), is one of the most critical motif in the development and production of modern real time system.

Having given an hint on concurrency we can now enter in a brief description of the two methods under examination.

4 FUNCTIONAL ORIENTED METHODS

As said above, in these methods, also referred as process oriented, the system representation is based on a collection of processes, each performing a function (or a set of functions) that is part of the overall purpose of the system itself. The processes operate on data running through the various elements of the system.

The mapping from the real world entities and the software design elements is not straightforward and such mapping is even less evident in the code structure. This can make difficult to understand, maintain and reuse such code.
On the other hand functional decomposition makes it

easy to cope with some peculiar aspects of real time systems like, for example, concurrency and timing requirements.

As a first example of a Functional Method we will briefly examine MASCOT, that stands for Modular Approach to Software Construction Operation and Test. The acronym itself identifies the salient features of the method. **Modular:** the key-point of the method is a particular formalism by means of which a complex software module may be broken down into a number of interacting smaller components; the process can, of course, be iterated until required to produce a manageable development unit. **Approach** is a synonyms of Method. **Construction:** the method incorporates the functions to build the software and ensure conformity among design, source code and object code in the target hardware.

MASCOT is suitable for the development of large distributed embedded systems. The emphasis is on the **large** in all its significances: large number of involved people, large number of requirements to be serviced simultaneously (concurrency) and large amount and assortment of hardware resources to be handled.
In general a system may be considered as consisting of a number of interconnected internal elements whose combined individual operations produce the overall effect of the system as a whole.
The MASCOT representation of a system is characterised by two basic types of components, the activities and the data areas called *IDAs* (Interconnected Data Areas). In a system the elements of the two types are interconnected to form a dataflow network, consisting of active elements (activities) that communicate through passive elements (IDAs).
Appropriate access mechanism are implemented to protect the integrity of the data and to ensure the propagation of the information.
In a network of concurrent processing no explicit time ordering is embedded, although a priority mechanism can be used at run time when necessary.

Another category of non-object oriented methods are the ones based on structured analysis and structured design techniques.
The first step on a typical structured system development is the analysis that deals with "what" a system must do. The result of this phase is a specification that should detail thoroughly, accurately, and consistently which functions the system shall implement.
There are various types of structured graphical forms to prepare a specification document, but they are basically similar. For example the Yourdon/De Marco variant uses three basic elements: Data Flow Diagram to provide a graphical representation of the system, the Data Dictionary to add written description of the data component of the system, and Process Specification which describes the system functions performed on these data.

Having specified "what" a system is supposed to do, our development should move into "how" it shall be implemented, that is the Design phase. As for any other (good) methods, structured design promotes foremost software design practices such as modularity, consistent interface definition and code reusability.
Modularity is the preminent answer to software design problems, specifically to complexity.
A graphical representation accomplished applying modularity rules makes the system easier to understand. Reducing module size also results in software units that are easy to code and test. Also changes are easier to control and implement while their consequence are more easily understood. Having discussed the Yourdon/De Marco method for defining system requirements, similarly we can consider the Constantine/De Marco structured design approach for software design. The design elements are simple, few in number and matching the Yourdon/De Marco ones: structure chart, data dictionary, and module specification. Their description is much the same of the basic elements of the Yourdon/De Marco.

5 OBJECT ORIENTED METHODS

Object Oriented Design (OOD) can be considered a "new" method for representing the real world in software.
In describing a system two main entities can be identified: the objects and the operations applied on those object. An object is a model of a real world entity, which combines data and operations on that data. If we considered our utility system example on Section 3, the engine, the turbine, and the gearbox are objects. The corresponding operations are the control of the oil pressure, temperature, and speed.

Many variant of the original OOD approach have been developed. One of the most popular is HOOD (Hierarchical Object Oriented Design). This method combines the traditional top-down approach with the benefit of an Object Oriented representation, allowing the introduction of a hierarchy among the objects in the design.
HOOD has been developed first time in France in 1987, specifically to support the design of software to be written in Ada. The first supporting tool has become available in 1988, year on which the method has been adopted by ESA for the Columbus project. In 1989 the new Version 3 of the HOOD Reference Manual has been produced. Other toolsets have been developed and HOOD has been adopted by space and industry users including, in 1990, the European Fighter Aircraft (EFA).
The HOOD design strategy is globally top-down and consists in a set of basic design steps, in which a given object (called parent) is decomposed in smaller components (child objects) which together provide the entire functionality of the parent object.

The process starts at top level with the root object, which represents the abstract model of the system, and terminates at the lower level where only terminal objects are present. Terminal objects are developed in detail and directly implemented into code.
A basic design step is in itself a small but complete life cycle. During the various phases of these cycles the software requirements are understood and restructured, an informal solution is outlined and described in terms of object at a high level of abstraction. Subsequently, child objects and associated operations are defined and a graphical representation of the solution is given by means of an HOOD diagram. Finally the solution is formalised through formal description of object and operation control structures. At the end of this phase the design structure may be automatically translated into Ada code.
The most crucial task in an HOOD design, and in general in any OOD variant, is the identification of the objects. In fact, while it can be intuitive to identify the operations it can be tricky to distinguish a consistent and appropriate set of objects.
The theory suggests that the designer firstly express the software requirements in a group of clear and precise definitions (in natural language) of the requirements themselves. From this text, nouns are identified as candidate objects, and verbs are identified as corresponding candidate operations.
To represent the dynamic behaviour of the system, that is a fundamental aspect of real time systems, Petri Nets and State Transition Diagrams can be used.
Among the main principles for identifying objects we can cite hardware devices to be represented, data to be stored and data to be transformed.

The intent of an object is to represent either a real world entity or a data structure. It should act as a black box hiding the data and allowing access only by means of operations. In this way the testing, debugging, and maintenance are eased.

6 FUNCTIONAL VERSUS OBJECT ORIENTED

This section is the essence of the paper and proposes the comparison between the two methods on different aspects, all relevant to a development of a typical large real time system for a Defence project.

6.1 General

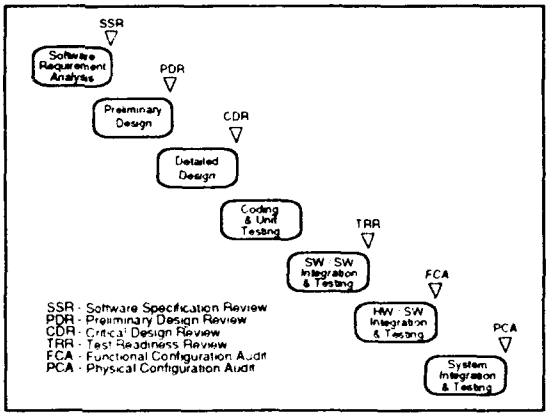
When assessing the suitability of a method to support software development at least two themes must be considered: the project constraints and the various aspects on which the comparison is to be performed.
Among the project constraints the main points to be considered are the programming language and the standards and procedure to be applied.
About the former we can note that since Mid Eighties DoD have requested Ada as programming language for their applications. As a consequence Ada has become a world-wide standard for virtually all defence project.
In considering standards and procedures we cannot forget DOD-STD-2167. This widely spread standard states the requirements for the implementation of a well defined and consistent software development cycle, the related monitor and control activities, and the relevant documentation. Even though each project has its own standards, they are usually derived from 2167 and must meet its requirements.
In our comparison we will consider a typical project whose standards meet the requirements of DOD-STD-2167 and adopt Ada as programming language. This situation is so widespread that we consider acceptable to limit to it our analysis.

The main aspects to be considered in comparing the methods are identified in the following list:
- Support to Life Cycle Phases
- Software Testing
- Software Safety
- Software Maintainability
- Relationship with DOD-STD-2167

6.2 Support to Life Cycle Phases

DOD-STD-2167(A) defines the classic "waterfall" cycle, the key point of which is the clear distinction among its various phases. Prerequisite for passing to a new phase is that the preceding is closed and its products validated at a Formal Review.
Main phases of this cycle are Software Requirement Analysis, Preliminary and Detailed Design, Coding and Host Testing, and Formal Testing.
Figure 3 shows the "waterfall" life-cycle as defined by DOD-STD-2167(A).

FIGURE 3: DOD-STD-2167 SOFTWARE LIFE-CYCLE



Requirement Analysis

A basic principle accepted by most parties is that whichever method is chosen it should be applied from the beginning. In case of the OOD that means that also the requirements should be expressed in an object oriented way.

This is because, as we have seen, functional decomposition methods localise the information around functions, while the object oriented localise it around objects. Several projects have learnt how this combination leads to overwhelming difficulty.
The assumption from above, that could be the solution as well, would then be that, when OOD is applied this should be done from the definition of requirements and the production of the related documentation.
However, this approach is not easy to follow. In fact, the first intuitive step in describing a system is to state the functions it shall accomplish. At least user requirements will always be functional oriented. It is possible, of course, to elaborate and express these requirements in on object oriented way, but the process will not be intuitive and smooth. Furthermore, to define the dynamic behaviour of a system, OOD requires the support of other methods like Petri Nets and State Transition Diagrams; this introduces additional and not fluent steps into the development process.
From these considerations it would appear that functional methods are stronger than object oriented methods in the early development phases, where errors are more costly than the ones introduced at later times.

Software Design

Software Design is divided in two steps, Preliminary and Detailed Design. The former is the definition of the overall software architecture that will be expanded and detailed in the latter.
The modern methods and (partially) tools have known big improvements in the last decade. It can, therefore, be assumed that when looking at Software Design as a stand alone set of activities, the support provided by the various methods can be considered comparable, at least in general terms. Advantages and disadvantages still exist, of course, and depend upon the characteristics of the specific project, but they usually compensate each other.
During a discussion held to assess the results of an evaluation exercise on both methods a developer said, may be a bit naively, that the main problem in assessing the two designs was to tell the differences between them. Surprisingly most of the attendees agreed.
In addition to the characteristics of the system under development, another aspect to be considered in our comparison is the support to the various phases of the design. In the preceding paragraph we have highlighted the hardness in applying object oriented methods in the requirement analysis and representation. This implies that there is very little chance for software developers to work on requirements genuinely object oriented.
From this consideration we can derive that, in general, functional oriented methods are stronger in the early stages of software design, that is going from requirements to top level design elements.

Coding

Another fundamental issue on any software development is the transition from design to implementation, in other words, the mapping of design elements into programming language constructs.
Before continuing our comparison we need a small digression on programming languages.
Ada was developed in the early Eighties to answer the challenge of the software crisis. It was not specifically manufactured to support object oriented design. Nevertheless, some of its features like data and processing abstraction, generic types, packages (that support information hiding), have revealed themselves particularly useful in implementing object oriented design.
Although, due to several real time deficiencies (some of which are expected to be corrected in the 9X revision), Ada is not considered the best choice on applications with very stringent real time constraints, its diffusion is such that several object oriented variants (e.g. HOOD) have been developed specifically to support it.
In the meantime, several object oriented languages are being developed, some of them with specific support for real time applications.
Therefore, from one side we have an object based language (Ada) and a set of OOD variants adapted to support it, and from the other object oriented languages specifically developed to support object oriented design.
Assuming we are adopting an object oriented language (Ada or other), we can certainly conclude that the transition from design to implementation is smoother if an object oriented design has been followed.

6.3 Testing

Software is a fundamental and costly activity in the software development cycle. An accepted figure is that up to the 40% of the total project effort lay on testing.
Two sets of inputs are fed into the testing process: the software configuration (Software Requirements, Software Design Documents, source code) and the test configuration comprising of test plans and procedures, test cases, and expected results.
It is important to note that the objective of software testing is to discover errors with the minimum amount of time and resources. A testing exercise should not be aimed merely to demonstrate that the software is error free.
To be considered successful software testing must discover errors in the software. As a consequence testing demonstrates that the software appears to be working according to specification.
Testing can be divided in two classes: **white box** and **black box** testing.

White box testing is a way to conduct testing to prove the correctness of the software structure, and that the internal functions perform according to specification. Black box testing, concentrate on the functional requirements of the software. It enables the tester to derive sets of input condition that should exhaustively exercise all functional requirements of a program.
A typical example of testing conducted applying the black box approach is the Formal Acceptance Testing conducted on the final software load.
Classic examples of white-box testing are the testing phases conducted "informally" on units and aggregate of units, also known as Unit Testing and Software Integration Testing.

A well accepted postulate is that exhaustive testing is impossible for large software systems. That means that no testing process will ever lead to a 100 percent correct program.
From this consideration we can evince that the key point for a successful software testing, and of the associated project, is modularity. Simpler and well built modules designed according to good engineering principles, are easier to understand and test. The tested modules are then integrated in more complex ones, on which different types of testing are performed. If the system has been correctly decomposed and the interfaces properly defined, that is the essence of modularity, then the software testing has good probability to achieve its objective.
Coming back to our comparison, we can deduce that the key for a successful software testing lays in an excellent design produced according to good engineering principles, with particular emphasis on modularity. Whether it is better that this design is developed according to a functional oriented or an object oriented methodology is a marginal and questionable argument. As usual pro and cons on different aspects compensate each other.

However, there is an aspect where the functional methods are definitely more appropriate. This is when, for whatever reason, a software load must achieve partial clearance. We have already seen that one of the main effects of the software crisis is the likelihood of software being late. In the light of this reasoning the possibility to deliver interim software releases, possibly complete but only partially tested, is one of the options to mitigate the impact on the entire project. The large number of parallel and highly integrated activities composing the development of a typical Aerospace product - a new aircraft for example - together with the fact that modern systems are full of nice-to-have functions, makes this option particularly viable and rather common. Think, for example, to preliminary software versions used on integration rigs or for on-ground aircraft testing.
In these cases functional methods are considerably better for the simple reason that partial clearance simply means to prove that a set of **functions,** considered essential for the purpose of the specific software delivery, are correct. This result is easier to achieve if the design of the system, in addition of being modular, is developed around functions rather than objects, as there should be fewer modules to be tested to clear each individual function.

6.4 Software Safety

The widespread diffusion of digital computer systems makes very common the situation of human life relying entirely on software. When this happens the software is commonly classified **safety critical.**
Specific methodologies, standards, and procedures must be applied in order to achieve the necessary confidence on the quality of such software making its development much more expensive (up to three times) and challenging than the average.
The problems related to the development of safety critical software are well known and are outside the scope of this paper. However, to continue our comparison, we must provide some hint on programming languages and their relationship with safety.
The aim of the High Order Languages (HOL) is to alleviate programmer's work-load in implementing composite set of actions with single code statements or providing useful but complicated facilities, a good example of the latter is Ada tasking. While this is certainly desirable from the productivity point of view, it can have disastrous effect on safety. In fact, one area of concern about software safety is the possible errors introduced by compilers. Any HOL statement is automatically translated into a number of simpler intermediate statements (their number depending on the original statement complexity) and thereafter into the object code. It is intuitive that the more complex the HOL statement the higher the probability to introduce errors during its implementation. Another argument for simplicity of the HOL is the need for a simple correspondence between the source code and its compiled version, to allow the correctness of the latter to be checked.

Due to their complexity HOL languages are therefore not particularly suitable for the development of safety critical software. Speaking of Ada, to overcome this problem different subsets of the original language have been developed. SPARK is one of the most popular, at least in Europe. The leading concept on which these subsets are based is to reduce the compiler complexity by restricting the use of particular language features. In SPARK the use of Ada **tasking** is excluded because of its high degree of non-determinism due to the extremely complex interactions between concurrent processes.
Exceptions are to be avoided because it is easier to write an exception-free program, and prove it to be so, than to prove that the corrective actions performed by the exception handler are appropriate under all possible conditions.

Generic units are to be avoided because the complexity introduced by them is not justified. Basically the problem is the difficulty to prove the correctness of all instantiations of a generic unit.
All Ada features requiring **dynamic storage allocation** are not allowed. This includes **access types, dynamically constrained arrays, discriminants and recursion.** Although to a different extent all these features make the problem of verifying compiled code impossibly difficult. Furthermore, dynamic storage allocation also makes usually impossible to establish memory requirements.
Scope, visibility, and overloading rules are remarkably complicated and confuse verification quite unnecessarily. To simplify this aspect **overloading, block statements, and use clause** are prohibited, while the use of **renaming declarations** is restricted.
A number of less important Ada features, which imply a penalty in complexity with no substantial benefits for programmers are also banned by SPARK.

From above we can see that most of the features that make Ada so attractive for implementing object oriented designs are forbidden, or highly undesirable, for safety critical applications. The same applies for any other HOL. From this we can derive that also OOD methods are not particularly convenient for safety critical software development. This doesn't mean their use is detrimental but simply that most of the claimed reasons that make OOD "the solution" for the software crisis, cease to exist in safety critical applications. In this field the "old good" functional methods can therefore still be considered the best choice. Of course, in a project involving the development of both safety critical and ordinary software the application of different methods can be impracticable. In this case the choice should be driven by the characteristics of the project itself.

6.5 Maintainability

Software maintenance is defined as the process to modify existing software leaving its main functions intact.
A reasonable percentage (less than 40%) of redesign and redevelopment of new code can still be considered maintenance. Above this figure we must speak of new development.
Software maintenance can take the form of software update or repair. Software update is when the functionality of the software is changed, usually because of changes to requirements or system architecture. Software repair leaves the original software functions intact.
Maintainability is the quality factor that indicates how easy it is to maintain the software, that is to modify it. It can be achieved by applying proper engineering criteria like self-descriptiveness consistency, semplicity, and modularity. The first three depend upon the programming language chosen (self-descriptiveness), the standards applied (consistency), and the working practice (simplicity). In any case they can be considered secondary if compared with modularity. They are, in fact, related to the description, rather than to the actual quality of the software.
Modularity is therefore the only property having straightforward impact on software maintainability.

Design methods and programming languages have severe impact on modularity. Although all modern methods enforce good engineering principles that include modularity, the leading principles of object oriented approaches (information hiding, abstraction, localisation) are essentially the same that form the basis of a modular design. OOD methods can therefore be considered inherently more appropriate to support modularity. This conclusion is strengthened by the fact that object oriented languages support modularity at a greater extent than any other HOL, and that the use of OOD methods in conjunction with these languages has proven to be quite advantageous.

6.6 Relationship with DOD-STD-2167

DOD-STD-2167 was originally issued in 1985 and is a framework document providing guidelines for the management and documentation covering the development of military guidance and control software. While the corresponding civil standard RTCA-DO/178 only addresses the certification of software as part of a system, DOD-STD-2167 covers the situation where the software is delivered as a stand-alone product.
To take into account comments received following the first period of application - and to resolve the major incompatibilities with the use of Ada and the progressing software engineering technologies - a new revision of 2167 was issued in 1988.

Some strict interpretations of the original DOD-STD-2167 requirements support functional decomposition rather than object oriented design. The 2167 "waterfall" life cycle model - with its stringent requirement with respect to beginnings and endings of each life cycle phases - is not particularly suitable for supporting OOD. The concept of "code a little, test a little" is difficult to apply.
The phase in which this aspect is most noticeable is the software design. DOD-STD-2167 splits it into Preliminary and Detailed Design, separated by the Preliminary Design Review (PDR). In OOD this separation is somehow artificial and introduces additional workload and in some cases can be deleterious.
This partial incompatibility of DOD-STD-2167 with object oriented can be explained with the fact that in 1985 OOD was not enough widespread to influence the well established confidence on traditional methods.

Nevertheless, having acknowledged the need to cope with the outstanding development of software engineering discipline, DOD-STD-2167A has relaxed some too stringent requirements and has become "method independent". This has conceded significant flexibility in preparing dedicated project standards, allowing to apply virtually all methodologies.
Therefore, the claimed incompatibility between object oriented approaches and military standards is not a problem any more.

7 CONCLUSION

For the benefit of those who are used to read only the Introduction and/or the Conclusion paragraphs of a paper we briefly summarise the outcome of the analysis performed on the chosen aspects of a typical software project.

- Functional oriented methods are stronger in the early development phases (**System and Software Requirements Analysis**) where errors tend to be more costly (See paragraph 6.2 Software Requirements).
- Functional oriented methods are stronger also in the early stages of **Software Design**, providing a better support to the transition from requirements to top level design elements (See paragraph 6.2 Software Design).
- Adopting an object oriented language (Ada or other), OOD methods ensure a smoother transition from design to implementation. (See paragraph 6.2 **Coding**).
- The support provided by the methods to the **testing** activities is comparable, but Functional methods prove to be considerably better to support **partial clearance** (See paragraph 6.3).
- In **safety critical applications** most of the reasons that according to OOD promoters make OOD "the solution" for the software crisis, cease to exist (See paragraph 6.4).
- OOD methods can be considered inherently more appropriate to support **maintainability**. (See paragraph 6.5).
- The incompatibility between Object Oriented approaches and military standards, primarily **DOD-STD-2167**, is not a problem any more (See paragraph 6.6).

Having analysed the primary aspects relevant to the development of a typical real time software system we are now in the position to propose a possible set of subjective answers to the questions put forth in the Introduction paragraph.
Based on my personal experience I believe that the answer to the first question is **NO**: in real time applications, OOD methods have not maintained the expectations. In the second half of Eighties OOD were considered the only viable option for future projects. Their shortcomings were justified with lack of experience, human innate resistance to change, and poor tool support. After 7-8 years the situation is basically the same. OOD is not spread as expected, the same objections are raised, and OOD promoters provide the same answer of 7-8 years ago.
The answer to the second question is **YES.** Functional methods can still support large real time system development. At least until somebody will be able to provide an alternative "magical solution".

To conclude the paper, I wish to summarise the outcome of the comparison in a few points:

i) Somehow Functional methods have allowed the Software Engineering community to survive - even though with severe problems - to the software crisis.

ii) The alternatives to Functional methods have not proven to be "the solution". Some of them may be comparable or even slightly better but not to the extent necessary to definitively remove the causes of the crisis.

iii) The main innovative and advantageous features of OOD methods are not fully exploited in real time systems.

iv) The design method and tool is only a part of the development environment. Programming languages, testing tools and strategy, Configuration Management tools and practice - complemented by appropriate standards and procedures - are other fundamental issues for any software project. These methods and tools cannot be considered in isolation and the choice of each of them must be based on solid and consistent technical arguments.

Finally, let me say the last word:

The existence of the software crisis cannot be denied and is due to tangible causes. Nevertheless, it has become an easy excuse to justify too many fiascos due instead to errors in understanding and managing the project needs, characteristics, and purpose.
At any rate for real time systems, as for any other applications, the key to success does not lay on Functions or on Objects, but in avoiding artificial and senseless solution based on purely commercial and political speculations. Nobody is so naive to think that politics and business can be ignored, but they must be somehow limited allowing enough space to apply rational Software Engineering principles and practices.

Discussion

Question C.L. BENJAMIN

In your presentation, you discussed the strenghs and weaknesses of functional methods and OOD methods. Do you recommend that work be done using both approaches? Is that possible?

Reply

I do not consider viable the solution of applying the 2 methods. I suspect that this will mean a sum of the shortcomings of both methods, giving no real advantage. I do believe that the choice of a consistent system development environment, complemented by appropriate standards and procedures, is the key for a sucessful project, irrespective of functions or objects.

Question K. ROSE

I have 2 questions regarding OOD & Ada, but first I have some comments. The first OO language was Simula 67, developped in Norway based on work at the Norwegian Defense Research Establishment in the early 60s, so OO is nothing new. Simula has inspired the development of Smalltalk and C++. Simula support concurrency but is not suited for Real Time because of inefficient memory management techniques. The ada designers, well familiar with Simula, for that reason, as stated in the rationale for the Ada design, chose not to make Ada an OOL. C++ has later proven that it is possible to develop an efficient OOL suitable for RT applications. I find it strange to perform an OOD and implementation in a functional language (Ada).

To what extent do you believe that your conclusions are influenced by Ada's limitations regarding both RT and OO? Do you believe that Ada 9X will solve Ada's problems regarding OO and concurrency/RT?

Reply

I stated that Ada is not an Object Oriented language, but an object-based language. I do believe, however, that several Ada features are more than desirable to support OOD. Indeed the use of Ada with OOD is widespread enough not to consider it "strange" to apply this approach. The proliferation of OOD variants to support Ada ebforces my opinion.

In answering the questions, I want to make a small remark concerning the so called "Ada limitations regarding OO". It is not posiible and convenient to hide the OOD limitations of OOD in real time systems development (aknowledged by the software engineering commnity) with claimed limitations of the HOL. The answer is then NO, the Ada limitations did not influence my conclusions, I do believe that OOD is not "the solution" to the software crisis on real time systems.

I am not familiar with Ada 9X, but the claims are that it will solve several Ada real time limitations. I did not hear anything about OO support, so I do'nt know.

Hierarchical Object Oriented Design - HOOD - Possibilities, Limitations and Challenges

by Patrice M Micouin
STERIA Méditerranée
ZI de Couperigne,
13127 VITROLLES, FRANCE

and Daniel J Ubeaud
Eurocopter France
BP 13
13725 MARIGNANE Cedex, FRANCE

SUMMARY

The goal of this article is to sketch a first evaluation after almost four year usage of the Hood methodology in the context of Ada real time software systems.
For this purpose, it is made up of four parts:
- First, we will give a brief outline of Hood methodology
- Secondly we quickly sketch out four years of Hood usage.
- Thirdly, we will summarize the main lessons learnt throughout this experience.
- Fourthly, we will outline some directions useful to follow-up in the future.

1 INTRODUCTION

The Hierarchical Object Oriented Design (HOOD) is an architectural (or preliminary) design method defined for the intention of the European Space Agency (ESA). This method is used in several space software development such as the launcher Ariane Vth, Columbus or the Spot 3th satellite.

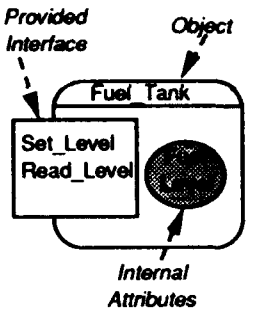
2 HOOD Paradigms

This part deals with the most significant features of the Hood method and its technical and institutional environment.

2.1 The Object Orientation

The Main pillar of the Hood method is its Object Orientation.
This statement means that the basic components of a software system aren't processes, functions, routines or program pieces but well-formed objects - ie consistent agregate of data and related operations referring to domain problem entities -.

For example, if an aircraft monitoring computer deals with the fuel level of the tank it isn't surprising that a Fuel_tank object appears in the design which holds the fuel level as an internal attribute and provides two services -set and read-.



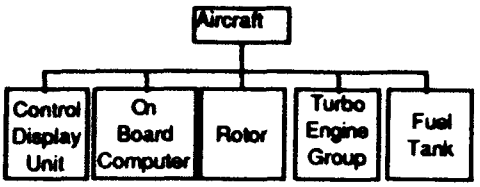
This orientation isn't a spineless accommodation with the mode buzzwords but the belief that object oriented software are more resistent to impact of evolutions and specification changes.

2.2 Top-Down Breaking-Down

The second pillar of the Hood method is its Top-Down Hierarchical Orientation.

For example, if a helicopter monitoring computer copes with different information concerning display units, rotor, engines , ... it may be interesting to hold this information in an upper abstraction "The_Aircraft" in order to manage the complexity.

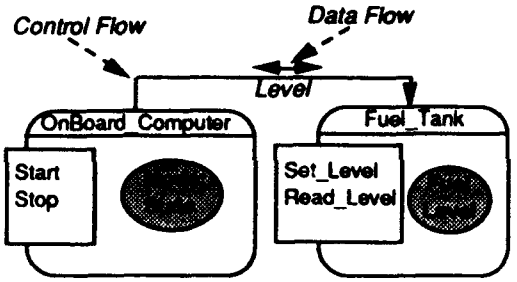
Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.



This second point is a very controversial one. Is Object orientation consistent with top-down approach? Does Object Oriented Design process have to be Bottom-Up oriented as B. Meyer and other Object "gurus" recommend it?

2.3 The Hood Notation

The HOOD method assumes that a software system may be designed as a collection of objects interrelated through two relationships:
The "*uses_services_of*" relation-ship and the "*is_composed_of*" relationship.



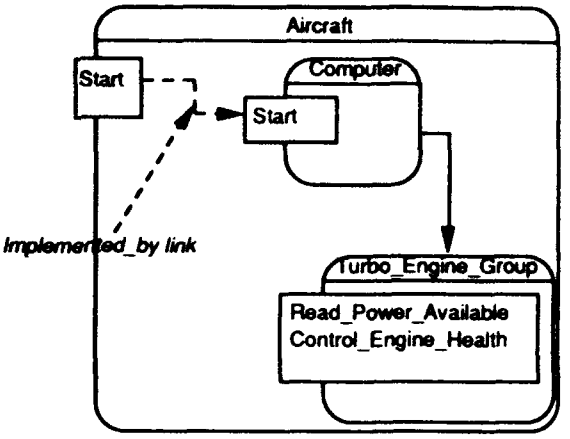
OnBoard_Computer uses services provided by Fuel_Tank

The "*use_services_provided_by*" relationship allows to model the action of an object (client) on an other object (server).

The "*is_made_up_of*" relationship allows to model probably the most basic cognitive mechanism

These two relationships provide a consistent frame to model large real-time systems satisfying reliability and operational long lifetime requirements.

Hood distinguishes two kinds of objects: Non Terminal and Terminal Objects. Obviously, Not Terminal objects have child objects. Every operation provided by a parent object must be implemented by a provided operation of a child object.



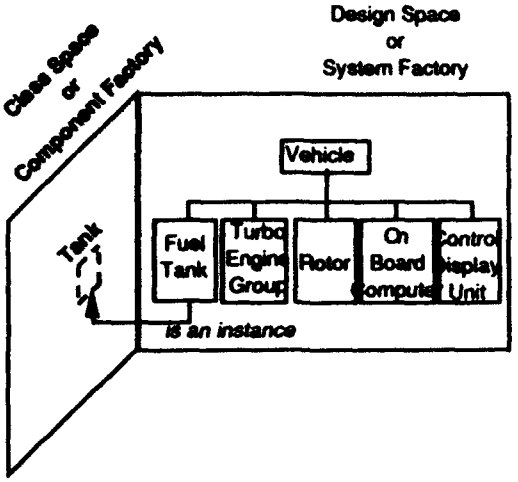
The Aircraft includes (or is the Parent of) the Computer and the Turbo_EngineGroup. The Turbo_Engine_Group is a child object. Aircraft.Start is implemented_by Computer.Start

2.4 System and Class Spaces

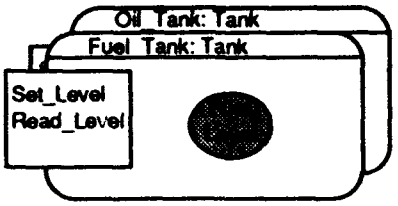
As opposed to Object Oriented Languages that mix classes and objects (instances) unseparatly, HOOD method splits distinctly software products according to two spaces: Design space and Class space.

On one hand, the Design Space holds operational software or parts of operational software.
On the other hand, the Class space collects reusable software components.

For example, if an helicopter monitoring computer copes with several objects designed in the same way such as Fuel_Tank and Oil_Tank it is interesting to catch the commonalities of these two objects into a reusable frame "Tank" located in the Class Space while instance objects named Fuel_Tank and Oil_Tank are located in the Design Space as Terminal Objects.



These two spaces are afterward linked by the instantiation mechanism.



In fact, Hood method gives here a practical consistency to the 1968-old Mac Ilroy [1] views about the software crisis and the industrial way to get out of the software proto-history.

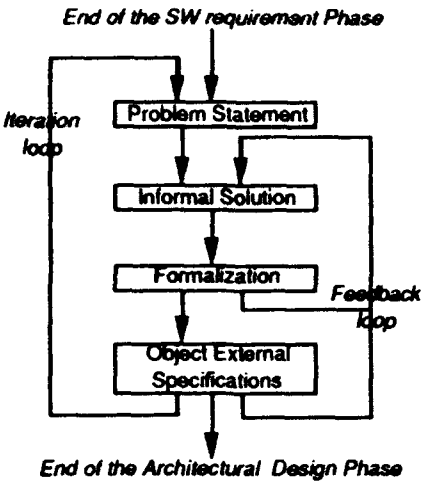
2.5 The Hood Process

The HOOD Method is not at all a notation (Poor method such as a method that is reduced to a notation).

The Hood method recommends an iterative process based on basic design steps.

This basic design step, inspired from Abbott [2], begins with the statement of the problem that the system (resp. the object) has to solve and ends with the external specifications of child objects.

Child objects may be non terminal object, and the basic design step resumes for these objects, or terminal objects. In this second case, terminal object will be definitively implemented (anonymous object) or instantiate from a class (instance object) during detail design and coding phases.



Here, Hood design process provides a technical consistent boundary between preliminary (or architectural) design and detail design. Preliminary Design deals with object identification and external specification of terminal objects while Detail Design deals with terminal objects implementation.

We consider that a terminal object generally corresponds to a package sized from 100 to 500 Ada lines.

2.6 The Hood User Group

Like Ada language is supported by the US DOD, the HOOD method is backed by ESA. Hood method is the property of its users joined in the Hood User Group (HUG).

The HUG collects user observations and defines method evolutions. The HUG garanties stability for Hood tool vendors concerning their investements.

The Hood method is described through Hood Manuals edited by the HUG.

The Hood Reference Manual.Its latest issue is the 3.1.1 issue released in july 1992 and co-published in about may 1993 by Prentice-Hall and Masson. And the Hood User Manual, its issue 3.1 revised at the moment by the HUG.

3 HOOD in Action

Hood method has been used on several navy [3] and space projects. This chapter presents Hood projects which are in the authors' scope.

3.1 Hood and the TIGER Helicopter Program

The TIGER helicopter is a German-French military helicopter under development with two main missions: Anti-Tank Mission (the Tiger :PAH2/HAC helicopter) and Support and Protection Mission (the Gerfaut: HAP helicopter). Eurocopter has the global responsibility of this development.

Concerning Avionics software, Eurocopter/France is in charge of the Mission avionic computers: The embedded software of Mission Computer and Symbol Generator (MCSG) concerning the PAH2/HAC mission and the software Armament Computer and Symbol Generator (ACSG/CDD) concerning the HAP mission.

3.1.1 The TIGER Mission Computers

MCSG and ACSG/CDD have been developped since 1990 at Marignane by joint teams of Eurocopter-France and Steria, as partner.
By 1990, After evaluation, HOOD has been selected as Preliminary Design Method. Four major versions have been released and the following are under development.
The size of the early versions is close to 50_000 Ada lines and the latest ones approach 70_000 Ada lines.
Time constraints are severe enough (time base: 20 milliseconds).
On the other hand, required reliability is high (Test programs multiply per 3 the number of Ada lines developped in the context of the MCSG and the ACSG/CDD).

3.1.2 Other Eurocopter Avionic Projects

Next to the MCSG and ACSG, Eurocopter France, in partnership with Steria, developes several other avionic software.

Some of them are in the frame of the Tiger program, such as

- Dolphin demonstrator for validation of AC3G which is a weapon control system (3rd Generation Anti-Tank missile) intended for the Tiger helicopter. This software involves 15_000 Ada lines.

- PUMA-PVS which is a demonstrator for validation of a Pilot Visionic System intended for the Tiger helicopters. This software involves 60_000 Ada lines.

Others are Eurocopter specific systems under development such as

- ACSR which is a rotor vibration control system. This software involves 10_000 Ada lines.
- ARMS which is a recording and monitoring system of helicopter health and usage. It experiments the Modular Avionics technology and will provide, in the future, maintenance on condition.
This software involves 22_000 Ada lines.
- PHL which is an engine monitoring system designed for light-weight and medium helicopters.
This software involves 20_000 Ada lines.

3.2 Other Projects managed by Steria

Independently, Steria developes other large Ada software with the same methodology.

3.2.1 Air Traffic Control System

In the Air Traffic Control Domain, Steria is in charge of the development of a subsystem of CAUTRA (Controle Automatique du Trafic Aerien). This subsystem monitors the traffic control system in its whole.This software involves 25_000 Ada lines.

3.2.2 SNLE Crew Training simulators

In order to train the SNLE submarine crews, the French DGA has designed several simulators (Soument Project).

Steria is in charge of the development of one of these simulators (Soument-SPP) using the Hood method. This software involves 45_000 Ada lines. Other simulators are developped with the Hood method.

Project	Domain	Size	Hood Tool	Host APSE	Target Compiler
SOUMENT-SPP	Simulator	35_000	Concerto	Vax-Ada	Vax-Ada
GDS	ATCS	25_000	STOOD	VERDIX/SUN	VERDIX/SUN
AIMS	SW engi-neering	45_000	STOOD	ALSYS/HP	ALSYS/HP
ACSG	Avionic	70_000	STOOD	RATIONAL	ALSYS/680X0
CDD	Avionic	20_000	STOOD	RATIONAL	ALSYS/680X0
MCSG	Avionic	70_000	STOOD	RATIONAL	ALSYS/680X0
ARMS	Avionic	22_000	STOOD	RATIONAL	ALSYS/680X0
AC3G	Avionic	15_000	STOOD	RATIONAL	ALSYS/680X0
ACSR	Avionic	10_000	STOOD	RATIONAL	ALSYS/680X0
PUMA-PVS	Avionic	60_000	STOOD	RATIONAL	ALSYS/680X0

4 Possibilities, Limitations and Desirables Improvements

4.1 Mastering Projects

It is generally difficult to determine the keys of any success. It depends on the teams, the development tools, the methodology and a lot of technical and human factors

With a background of almost four years, we could state that Hood methodology assits project managers efficiently in their jobs. We have been able to release complex sofware with the required quality on time.

This result has been reached with different teams, different Ada and Hood tools and within different domains.

Productivity figures depend directly on
- the complexity of the production line (Host, Emulation environment, Software Test Bench).
- the required safety and test effort

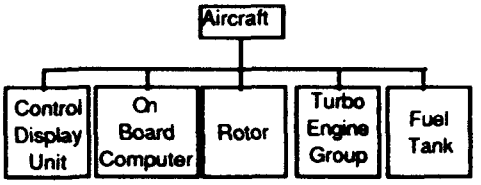
Concerning GDS project, productivity figures are over 50 Ada line per day from preliminary design to integration.

4.2 Mastering Complexity

The main lesson that we have learnt from our experience is that the Hood method is very powerful means to master complexity.

In spite of raised objections and true undesirable secondary effects, we think that this power is due to the top down hierarchical orientation of Hood.

This aspect has been a key factor to master the complexity.



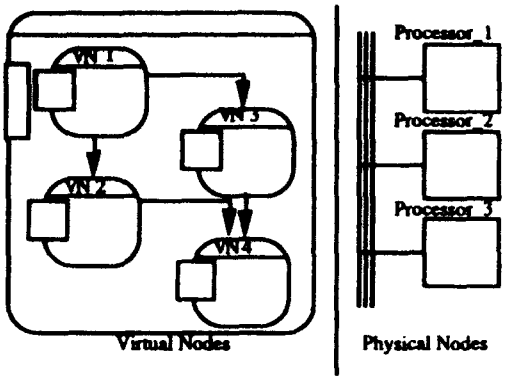
After Interfaces and behaviour are specified, each object may be independently designed.

We have been able to put subteams on different parts of design allowing parallel developments.

4.3 Dealing with Distribution

In order to deal with multi-processor architectures and distribution problems, Hood has introduced the Virtual Node concept. A virtual Node is a software piece able to be allocate to a processor. So, a distributed software system is a virtual node network which has to meet a hardware system that is a physical node network.

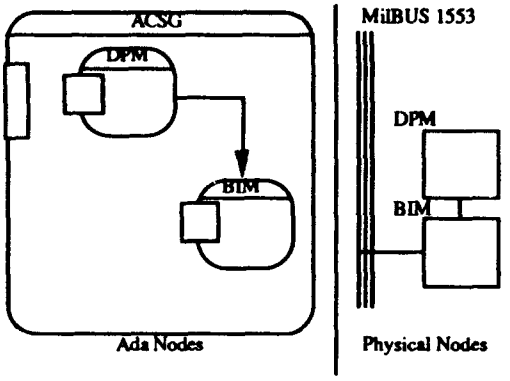
Unfortunatly,this approach which is very attractive is not really efficient In fact, inter process exchanges are very sensitive to the hardware architecture. Local and remote exchanges do not have the same performances and temporal behaviour depends on processor allocation.



For example, the temporal behaviour of the system may be very different if NV1 and NV2 are put together or not on the processor..

More, Virtual Node has no specific equivalent in Ada83.

So, we have discarded, for the moment, the use of Virtual Nodes about the design of multi-processor applications. (Tiger and Gerfaut computers are multi-processor computers). We are waiting for Ada9X partitions.



For example, ACSG software is designed as two Ada Nodes BIM (Bus Interface Module) and DPM (Data Processing Module) mapped in two physical processors. DPM and BIM Role, synchronisations and communications between Ada programs BIM and DPM are fully specified before the design.

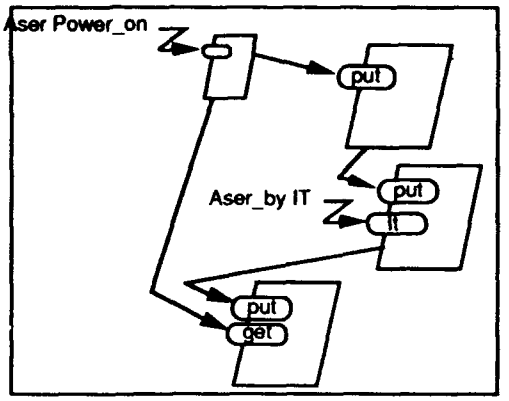
4.4 Dealing with Real-Time Aspects

Concerning Real-Time aspects, the Hood method provides several features such as active objects which deal with parallel evolutions and constrained operations which deal with cooperation between active objects.

But it is obvious that these features are insufficient in managing dynamic views of the solution and Hood is not fully real time orientated.

Our experience shows this is not really a major drawback. In fact, it is possible to add some guidelines such as recommandations provided by H Gomaa [4], R.J.A. Buhr [5] or Nielsen and Shumate [6] and a dynamic behaviour formalism desciption such as SDL[7], GRAFCET[8] or STATECHARTS [9]. These useful additions are compliant with the process frame recommended by Hood and may be added into the Hood User Manual.

For example, at the processor level, we have introduced a specific activity related to the management of asynchroneous events and time constraint services.



The goal of this activity is to design the real time architecture of the software.

The processor is a shared resource for tasks, so the processor level is the right level to tackle tasking architecture. At lower levels this activity would be too late and without global vision.

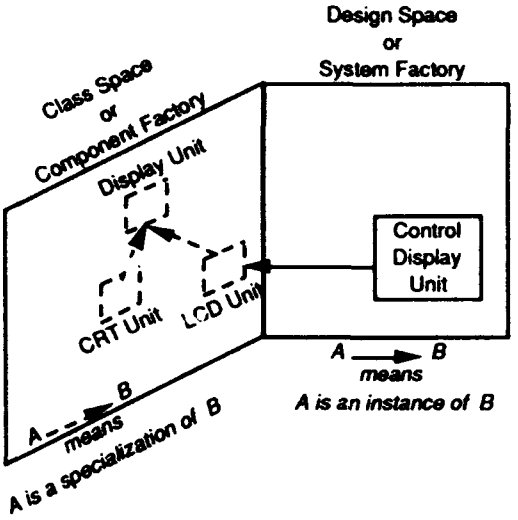
This activity aims to define:

An on-going study pointing out these difficulties has been ordered by ESA from a team led by STERIA.

5 Challenges

5.1 Hood and Inheritance

As already described, Class space is the reusability context. Presently, this reusability is fully based on the abstract data type and the genericity concepts inherited from Ada.



Is inheritance a key concept to reusability? Probably, yes.
Unfortunately, due to its origin as a Ada Design method, Hood method does not deal with inheritance.
Despite its capabilities, this lack may become a handicap in the future.
Several ways are forseen. Our feeling is that the Hood way towards inheritance must not imitate Object Language facilities but maintain a strict separation of concerns. - No change in the Design Space which is the system maker workshop.
- Including a "Generalization/ Specialization" relationship at the Class Space level conceived as a laboratory where software components are elaborated, classified and improved. Class space is, with this policy, a reusable software component repository.

5.2 Hood and non Ada Target Languages

Hood is a design method and it is not desirable that its fate should be bound to a particular programming language. We are conviced that Ada is the language best adapted to programming in large.
But someone may have another opinion and Hood must be able to address other programming languages, C++ for example.

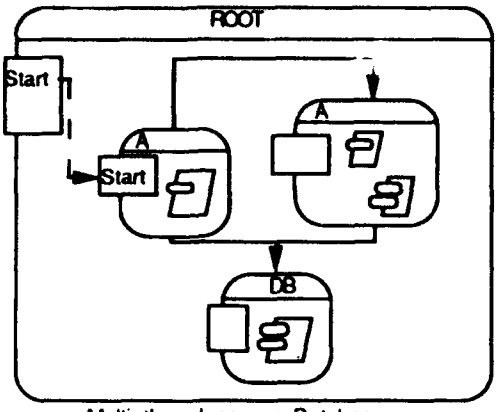
5.3 Dynamic Description

Probably, Hood needs a normalized way, validated by HRM rules and HUM guidelines, to describe dynamic behaviour of software systems and objects. Our feeling is this formalism must be graphical and easy to use. SDL, Grafcet or Statecharts are, in our opinion, are good candidates. Other authors recommend Petri Nets [12].
And certainly, the main difficulty about this issue is reaching a consensus between Hood users who are generally a very imaginative population.

---

[1] Mac Ilroy, Mass Produced Software Components, 1968 NATO Conference on Software Engineering.
[2] R.J Abbott: Program Design by Informal English Descriptions, Com ACM, Vol 26 N° 11
[3] M. Lai, An overview of several French Navy Projects, Ada Europe, Dublin 1990.
[4] H Gomaa: A Software Design Method for Real Time Systems, Com ACM, Vol 27 N° 9
[5] R.J.A. Buhr:System Design with Ada
[6] Nielsen and Shumate, Designing Large Real Time Systems with Ada
[7] SDL: Specification and Design Language CCITT Z101 to Z104.
[8] GREPA, Le GRAFCET, de nouveaux concepts.
[9] STATECHARTS, D. Harel, StateCharts: A visual formalism for complex systems, Science of computer programming 1987.
[10] Lui Sha, and John.B. Goodenough, Real-Time Scheduling Theory and Ada,Computer, April 90
[11] J. Poudret, Hood and DOD STD 2167A, Hood User Group Meeting Pisa, 3 April 92.
[12] Labreuille et alii, Approche Orientée Objet HOOD et Réseaux de Petri pour la conception de logiciels temps réel. Toulouse 1989.

- the optimum number of tasks necessary for fitting temporal and functional requirements, hardware constraints and the task priorities.
- the type of these tasks (periodic, aperiodic, sporadic, ...).
- the interface of these tasks (synchronisation, communication).
- the priority of these tasks.

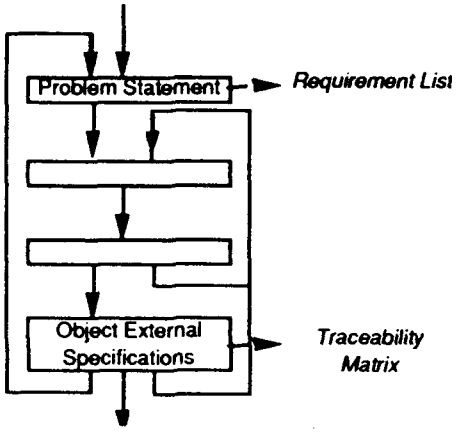


Multi_thread_access_Database

This tasking architecture is afterwards casted on the object architecture. This activity allows a true management of time constraints and is consistent with the Rate Monotonic Analysis [10].

4.5 Hood and Traceability

The Hood process suggests a very simple and efficient mean to maintain the traceability of requirements through the design process.



At every basic design step a requirement list may be allocated. These requirements are the requirements that the object under design has to satisfy.

Requirement List	Child Objects				
	Obj1	Obj2	Obj3	Obj4	Obj5
Req_n_m	X	X			X
Req_i_j		X	X		
Req_k_l	X	X			X
Req_n_j		X		X	
Req_i_m		X			

After the child object break down a traceability matrix which states the contribution of each child to the satisfication of each requirement.

4.6 Hood and Documentation

Providing a design documentation such as DOD 2167 PSDD and SDD which is readable and consistent with Ada sources is a requirement that may lead to extreme effort.
Hood tools provide documentation facilities. Used without predefined strategy they provide boring, unreadable and endless reports. In fact, these documentation facilities need associated guidelines in order to produce a readable and efficient documentation [11].

A lot of information collected during the Hood design process such as "informal strategy of solution" or "Design Choice Justifications" do not have an equivalent in DOD PSDD and state Hood documentation as a maintenance oriented documentation

4.6 Hood, Ada Code Extraction and Maintenance

As Hood design ouput. Hood tools may provide Ada Skeleton Architecture consistent with Ada extraction rules. Generated Ada code is run time efficient and globaly constitent with Hood principles.

But the lack of integration between APSEs and Hood tools has caused some heaviness concerning the maintenance phase.

Object Oriented Design of the Autonomous Fixtaking Management System

Joseph Diemunsch
WL/AAAS-3
Wright-Patterson AFB OH 45433
USA

John Hancock
TASC
55 Walkers Brook Drive
Reading MA 01867
USA

1. BACKGROUND

The Air Force Avionics Laboratory has sponsored several efforts to increase the accuracy of aircraft navigation functions while decreasing crew workload through the application of intelligent systems. Two such efforts were the Adaptive Tactical Navigation (ATN) System and the Autonomous Fixtaking Management (AFM) system, which were both awarded to The Analytic Sciences Corporation (TASC). An intelligent system to aid the pilot with navigation functions was developed under the ATN program. This system incorporated real-time knowledge base software to manage the tactical navigation moding, fault tolerance, and pilot aiding to provide a robust navigation prototype for the next generation fighter aircraft. The ATN program highlighted the aircraft weapons officer's heavy workload associated with the location and identification of fixpoints to update and verify the accuracy of the navigation system. With this problem in mind, it was determined that an intelligent system was needed to automatically locate, image, and identify fixpoints and update the navigation solution.

The AFM System was developed to prove the feasibility of automated navigation updates using tactical sensors and existing mission data processing systems. Several technologies developed under ATN were incorporated into the AFM system including a proven simulation of the navigation sensors, controllers, and mission planning and management software. Automation of human fixtaking activity required integration of several emerging technologies including a real-time data fusion architecture, neural network and heuristic automatic recognition algorithms, and associative memories to retrieve fixpoints from on-board databases. Integration of these diverse technologies was simplified by the employment of an object-oriented software development approach and real-time control system.

2. INTRODUCTION

The Autonomous Fixtaking Management (AFM) program's goal was to develop and demonstrate an automated aircraft navigation system which would not only reduce the crew navigation workload burden, but could also be used as a backup to the Global Positioning System (GPS). The AFM system matched fixpoint images from on-board databases to imagery acquired through Synthetic Aperture Radar (SAR) and Electro-Optical (EO) sensors to determine the vehicle's position. The AFM system eliminates the requirement for the workload intensive process of manual fixtaking. This was accomplished by automating the activity of a tactical navigator in selecting, imaging, and interpreting ground-based features and associating them with reference source data to derive navigation updates. Development of a real-time AFM system ensured mission success by maintaining an accurate navigation solution without increased crew workload.

The AFM system used a Hyper-Velocity Vehicle (HVV) for it's baseline study. A HVV is generally considered to be a vehicle which can exceed five times the speed of sound, or mach 5. The vehicle under consideration in the AFM program is assumed to be capable of rapid, short-notice, conventional takeoff, climb to endoatmospheric cruise, and if required, insertion into low earth orbit. The goal of the AFM system was to integrate tactical sensors, processing, mission data, and map databases which existed in the design of a potential HVV. The mission of the HVV, for the purposes of this program, was high accuracy and rapid response reconnaissance at long distances from the launch location.

The exceptionally high speed, and correspondingly short mission time, associated with an HVV have the effect of increasing crew workload. In the event of GPS failure, the mission oriented workload is increased due to a decrease in time available for setup, fixpoint acquisition, and navigation correction. As supported by the findings of aircraft cockpit automation studies such as the Air Force's Pilot's Associate Program, the greatest mission effectiveness payoffs are obtained by providing the crew with information that enhances situational awareness while automating system management tasks required to produce the information. Reliable, accurate navigation is key to crew situation awareness and HVV mission success. Automating related management and operational tasks such as data consistency monitoring, mode planning/switching and sensor image interpretation is a significant opportunity to improve mission effectiveness. Although the HVV was used as the baseline vehicle it

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

is easy to see how the technologies involved and system design can be used for a variety of missions and airframes. The concepts are just as applicable to a tactical mission on any aircraft.

This paper will focus on the software engineering techniques utilized to implement the AFM system. The AFM system used modular software design and object oriented development techniques to integrate models of existing on-board sensors, processing, mission data, and map databases. System requirements were developed by applying an in-depth knowledge of mission requirements and real-time intelligent avionics. Several diverse technology disciplines were integrated including neural networks, associative memories and real-time data fusion architectures to develop an effective and technologically advanced system. Neural networks along with other automatic target recognition techniques were used to find the fixpoints from the SAR and EO images. Associative memories provided real-time retrieval of the fixpoints from on-board databases. Finally, the activation framework architecture, a real-time object-oriented data fusion architecture, was used to integrate the overall system and provide a software engineering methodology for sensor management.

3. OVERALL SYSTEM DESIGN

The overall system design integrated several advanced technologies into a real-time simulation of the AFM system as illustrated in Figure 3-1. The system design was developed from the following conceptual model. First, a mission plan would be given to the pilot and possibly loaded onto the aircraft. The mission plan provides information such as route planning, environmental data, and target information. The real-time simulation required flexible symbolic logic to intelligently utilize the mission planning and in-flight mission data. Embedded within the mission plan is navigation data, which the simulation extracts to develop a navigation plan. This navigation plan consists of navigation modes and determines the aircraft position and times to send navigation updates to the mission manager. In this simulation a-priori planning information was used and in-flight replanning was done in the real-time navigation planning/monitoring module. As the mission path is traversed fixpoints are located with simulated sensors and compared to fixpoints retrieved from on-board terrain and feature databases. This comparison provides a navigation offset which is fed back to the navigation simulation. The large fixpoint database search required efficient storage methods and the image interpretation required parallel processing technology to achieve real-time

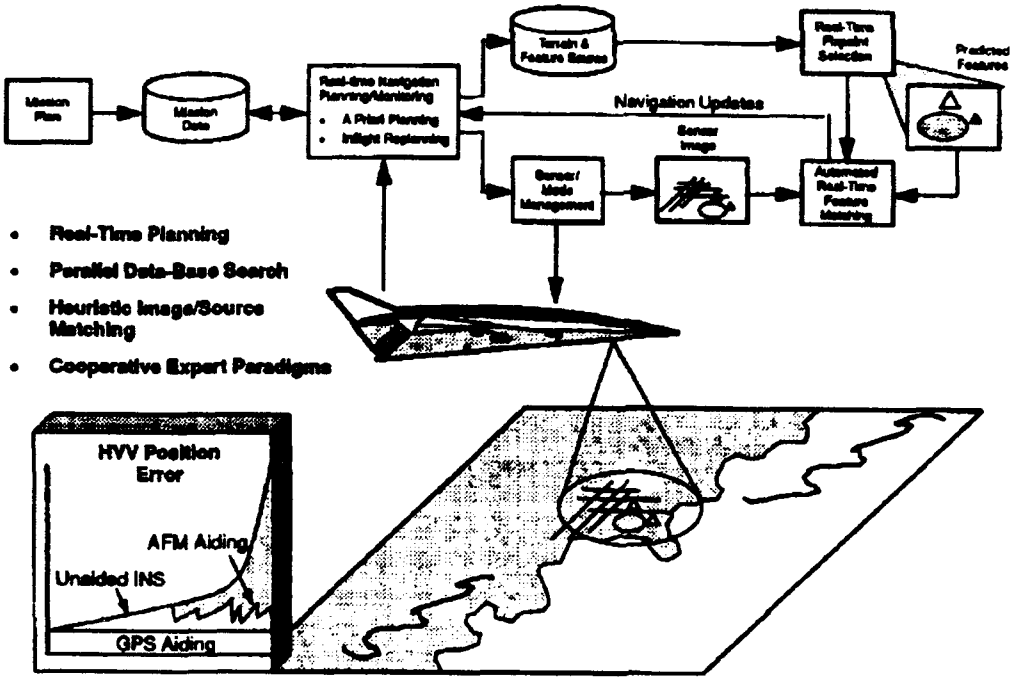


Figure 3-1. AFM System Design Overview

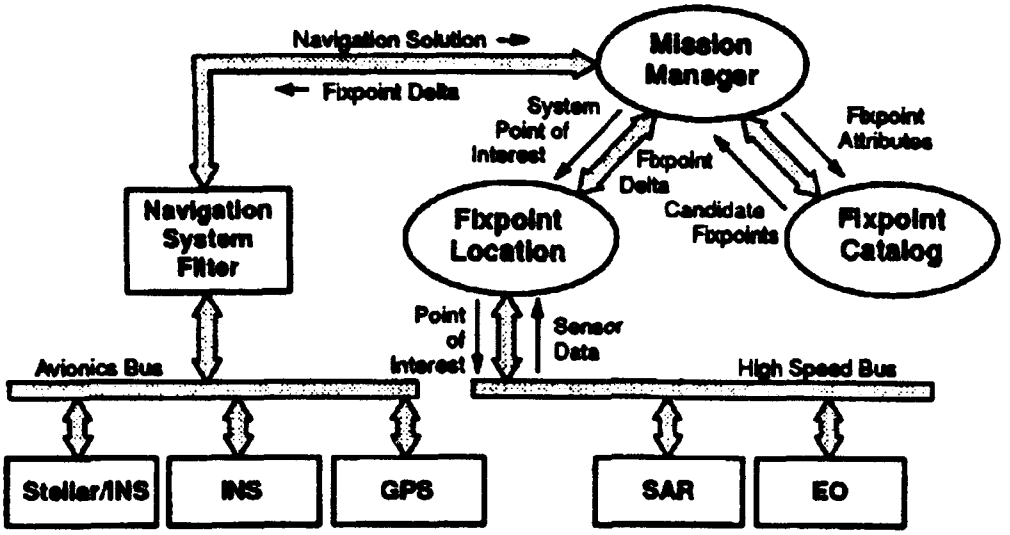


Figure 3-2. AFM High Level System Design

performance. The AFM system design applied associative memories, artificial neural networks, and AI planning technology to meet the real-time system requirements.

The design approach used for the AFM system separated the simulation into four major sections; the Mission Manager, Fixpoint Locator, Fixpoint Catalog, and the Navigation System Filter as illustrated in Figure 3-2. The majority of the Navigation System Filter was developed under the ATN program and will not be discussed in detail for this paper. The AFM system also relied on an intelligent data-fusion algorithm partitioning and system development approach developed under ATN. The approach to mission management is based on real-time coordination of a variety of cooperating agent processes orchestrated by an intelligent data fusion agent; the Mission Manager. Two critical agents which are unique to AFM are the Fixpoint Catalog and Fixpoint Locator. The Fixpoint Catalog employs a parallel processing associative memory technology to solve the problem of selecting a candidate fixpoint set from large databases of possible relevant terrain features, in real-time. The Fixpoint Locator utilizes intelligent feature classification technology to automate the fixpoint recognition activity of a human navigator.

The overall system design focused on developing, maintaining, and utilizing a dynamic mission plan to determine navigation moding along the planned flight path. The AFM design is based on a hierarchical decomposition of the mission planning and data interpretation functions. The three major agents in the AFM system required diverse computational approaches to meet the real-time performance requirements of the HVV environment. An object-oriented design and integration approach was used to facilitate agent paradigm encapsulation, provide consistent inter-agent communication, and enable real-time control prioritization.

At the lower level of the hierarchical design, individual software modules at the agent level were designed to communicate using the same message passing paradigm. The module design benefited from the algorithm encapsulation which allowed independent development of agent modules. The consistent inter-module communication interfaces allowed concurrent module and agent development by independent designers.

4. MISSION MANAGER

The overall controller of the system is the Mission Manager. The Mission Manager agent determines when to retrieve upcoming fixpoints from an on-board data base, commands SAR and/or EO sensors to image fixpoints, and orchestrates correlation of these fixpoints to update the navigation solution. The Mission Manager is the executive over the objects and assures cooperative communication and interaction. The underlying architecture allowing the communication structure is the Activation Framework (AF) paradigm.

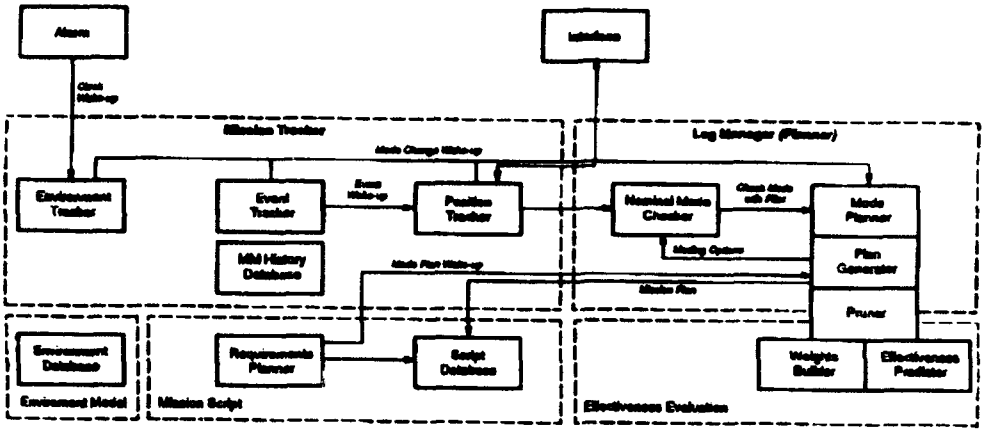


Figure 4-1. Mission Manager

4.1 MISSION MANAGER DESIGN

The Mission Manager design plans navigation resource utilization on a mission subleg basis, schedules fixpoint updates to the navigation filter, and monitors mission navigation performance along the planned mission route. These functions are performed by the Mission Manager utilizing five major components: the Mission Tracker, the Leg Manager, the Environmental Model, the Mission Script, and the Effectiveness Evaluation as shown in Figure 4-1.

The Mission Tracker monitors input reports of the mission progress including the mission position and external environment. The Leg Manager uses inputs from other Mission Manager modules to plan, implement, confirm, and maintain navigation system modes and functionality. The Environment Model contains a-priori models of the HVV, the sensor conditions, and atmospheric interactions for intelligent sensor selection. The Mission Script contains information on the mission route, targeting, and emission control over the route. The Effectiveness Evaluation module monitors fixtaking activity to ensure that fixes are valid. Within the Effectiveness Evaluation module, the Effectiveness Predictor relies on the built-in checking in the Fixpoint Catalog and Fixpoint Locator to assure accurate operation at the data interpretation levels. The Mission Script and Environment Model contain pre-mission planning data to support the AFM systems in-flight planning and analysis.

A symbolic mission script database is utilized by the Mission Manager to maintain mission goals, requirements, and restrictions as well as to schedule events (e.g., fixpoint updates). The Mission Script is a result of pre-mission planning and is stored in the script database. This script database consists of a series of leg intervals between way points, which is used by the Requirements Planner to setup the modes for the way points. The Mode Planner then utilizes these legs to determine where planned navigation mode changes are needed. The integration of all data and activity is controlled by the Fixtaking Planner.

The mission plan is stored in the Script Database as a tree structure. The root of the tree represents the entire mission. The second level of the tree shows a series of mission phases, each with predefined in-flight requirements. The third level of the tree consists of all the legs in the order in which they occur. Some legs have further branches which end in leaves representing subleg intervals. Traversal of the tree along the bottom leaves corresponds to the schedule of navigation moding, events, and environment along the mission route.

A list of properties characterizing the corresponding intervals are represented at each node in the tree. The properties include the endpoints of the interval, the planned nominal mode, fixpoint locations, GPS satellite visibility, and a list of requirements for the interval. The requirements are the navigation accuracy on the segment, the maximum susceptibility to Electronic Counter Measures (ECM) that can be tolerated, and electronic radiation restrictions. These requirements may be refined by the Mission Manager.

Along each leg of the mission route, the Mode Planner and Mode Complex objects perform constraint-based planning for the optimal navigation

mode and preferred fixpoint locations. Planning is based on a-priori mission requirements, current navigation mode, and physical conditions. Constraints on choice of sensors and fix timing originate at initial mission download and at asynchronous event times. Mission replanning becomes necessary when the current navigation mode is determined unhealthy, a sensor fails, or the expected ECM environment changes.

The low level structures such as the clock object, messages, planning structures, mission scripts, and AF communication facilities are object-based code. Object-based structures, such as messages, encapsulated data and code, provide flexible input/output entities, which localize translation mechanisms and reduce coding errors. The main structure is provided by the AF and the modules under this architecture called Activation Framework Objects (AFOs), these concepts are further explained in the section below. The AF, AFOs, scripts, all message types, message bodies, and other agent specific objects are based on a root object class which specifies default access utilities and operators. The default object capabilities include: print, get, put, stream operators, logical operators, and other fundamental functions, and data. Where required, specific derived objects redefine default access functions and operators.

Message object pointers are the root object type and can be decoded using the built in data available through an inherited parameter. The general pointer type for many objects combined with the type identifier provide the capability to store the dissimilar objects in the same data structures. The homogeneous storage mechanism allows for compact list and object storage. The simulation and planning code is simplified as a result of the compact, unified storage representation.

The Fixtaking Planner design interface is illustrated in Figure 4-2. The Fixtaking Planner links the Mission Manager to the AFM subsystems, the AFM Fixpoint Catalog, and Fixpoint Locator. The Fixtaking Planner is a control focus in the AFM architecture, orchestrated in an AFO under the Mission Manager. All communication is conducted through the AF facilities. The Fixtaking Planner AFO uses messages to communicate with the Effectiveness Prediction, Mode Complex, and Event Tracker Mission Manager AFOs.

The Fixtaking Planner uses the interfaces illustrated in Figure 4-2 to orchestrate fixpoint updates and navigation mode changes. When a plan is created or a replanning event is completed the Fixtaking Planner sends a list of times to the Alarm Clock. When a scheduled wake up time is reached the Alarm Clock object activates the Event Tracker which sends a message to the Fixtaking Planner. If the Event Tracker detects that the plan is nearing the time to take a navigation update, the Event Tracker notifies the Fixtaking Planner.

The Fixtaking Planner estimates the position of the HVV at the time of the planned fix and uses specific calculations to determine a point on the ground which will be visible to that sensor. The visible point is sent to the Fixpoint Catalog as a Candidate Fixpoint. The Fixpoint Catalog locates a known fixpoint for the sensor type located near the candidate point and returns that fixpoint to the Fixtaking Planner.

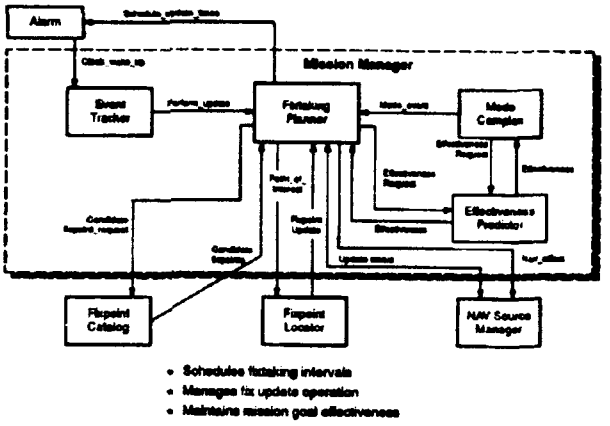


Figure 4-2. Fixtaking Planner AFO Interfaces

When the HVV is near the area of the fixpoint, the Fixtaking Planner sends the candidate fixpoint to the Fixpoint Locator which images the area, locates the fixpoint in the image, and returns a navigation offset. The Fixtaking Planner checks validity of the navigation update before sending the update to the navigation filter.

4.2 ACTIVATION FRAMEWORK ARCHITECTURE

The Activation Framework (AF) concept is based on an approach to real-time object-oriented implementation known as Communicating Expert Objects (CEO). In the CEO approach hypotheses are distributed among expert objects that communicate with each other by exchanging messages as shown in Figure 4-3. Procedures within an ATN or AFM expert object contain both factual hypothesis knowledge and procedural knowledge relating to its functional domain (e.g., mission management).

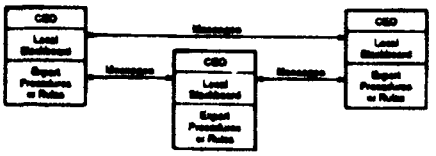


Figure 4-3. Communicating Expert Objects Architecture

An AF forms a community of AFOs (Activation Frame Objects) as shown in Figure 4-4. Each AFO is an expert in a limited problem domain and is the guardian of a set of private hypotheses. Each AF is a process which creates the environment in which all of its AFOs execute. Multiple AFs might coexist on the same processor or on multiple processors connected by a network, as shown in Figure 4-5.

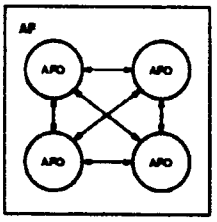


Figure 4-4. Activation Framework Concept

An AF is equivalent to a process executing within an operating system. Communication between AFOs is provided by AF services using a message passing mechanism. Message passing among AFOs is implemented by operating system level message passing mechanisms including, in the case of multiple processors, network protocol processing.

The flow of control within an AF is shown in Figure 4-6. The scheduler selects the next AFO to be activated. The procedural code of the activated AFO is then executed. The AFO can then use different AF services (typically message sending and message receiving) during the execution of the procedural code. When the AFO returns control, the messages sent during its execution are actually delivered to the receiving AFOs.

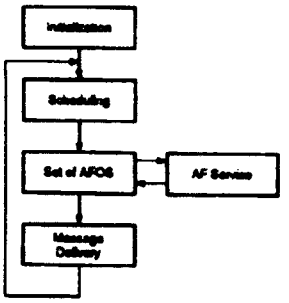


Figure 4-6. Flow of Control Within an AF

Each AFO has an input message queue and an output message queue. Message sending and receiving is depicted in Figure 4-7. When an AFO wants to send a message, the message is put on its output message queue by an AF service. When an AFO wants to receive a message, the message is taken off the AFO's input message queue and made available by an AF service. The actual passing of the message from originating AFO to destination AFO (either within or outside the AF) is done by the delivery procedure after the AFO returns control to its governing AF.

In the current scheduling scheme, each message is provided with a measure of its importance, the message activation level. Each AFO has an AFO activation level and an AFO activation threshold is

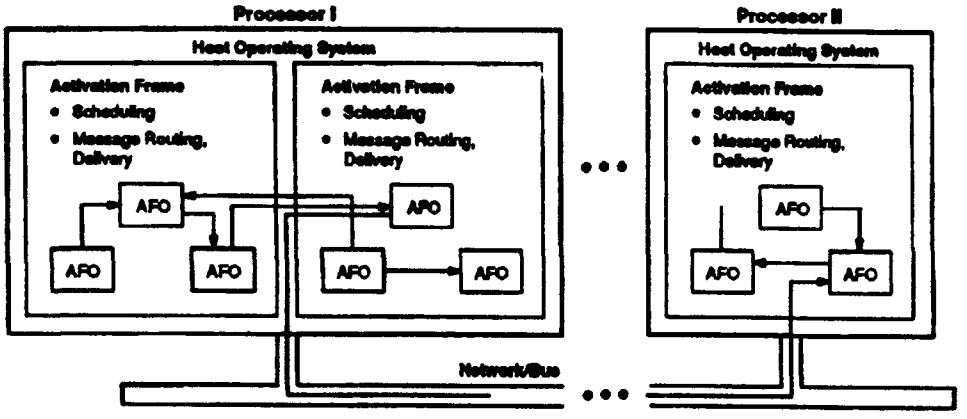


Figure 4-5. Activation Framework Concept

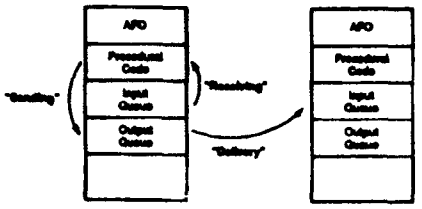


Figure 4-7. Message Passing by the AF

used by the scheduler to determine which AFO is next to execute. The AFO's activation level is the sum of all the message activation levels on its input message queue. An AF schedules its AFOs for execution based on the difference between their activation levels and activation thresholds. The AFO whose activation level exceeds its threshold by the greatest amount is executed next. All messages on that AFO's queue are then serviced.

The advantage of the AF architecture is that unlike global blackboard structures the scheduling mechanism is distributed throughout the system. This distributed scheduler removes the bottleneck of all processes going to one centralized scheduler, thus allowing the achievement of real-time performance. Another advantage the encapsulation mechanism provides is that symbolic as well as numerical processes can be run under a common framework.

5. FIXPOINT CATALOG

The high speed of the HVV limits the temporal window of opportunity on fixpoints. At 100,000 ft altitude mach 6 flight, the time from when a fixpoint appears on the forward horizon until it disappears into the aft horizon is approximately 40 seconds. This time limitation requires preselection of candidate fixpoints prior to arriving at that location in the mission. The Fixpoint Catalog uses a preferred look location from the Mission Manager to retrieve a fixpoint of the appropriate type, (e.g., SAR, EO) in a fixed time period. The output from the Fixpoint Catalog is used by the Fixpoint Locator to obtain a sensor image of the appropriate ground region and to locate the predetermined features within that image.

5.1 OVERALL FIXPOINT CATALOG AFO DESIGN

The Fixpoint Catalog associative memory algorithm described below is encapsulated in an AFO. The AFO will handle communication between the Fixpoint Catalog, the Mission Manager, and the Fixpoint Locator. In the initial prototype this AFO loaded geographically relevant groups of fixpoints into the storage matrix. The storage matrix loads are file I/O performed sequentially as the mission progresses.

5.2 ASSOCIATIVE MEMORY TECHNOLOGY

To facilitate rapid, constant-time candidate fixpoint retrieval, an associative memory approach is employed in the fixpoint catalog. The fundamental approach is illustrated in Figure 5-1. The fixpoint attributes are encoded as a vector, and run through an iterative, simulated, parallel search of the fixpoints stored in the associative memory.

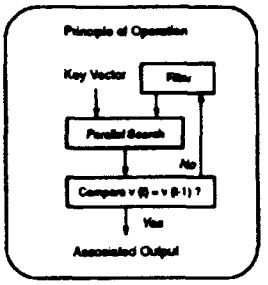
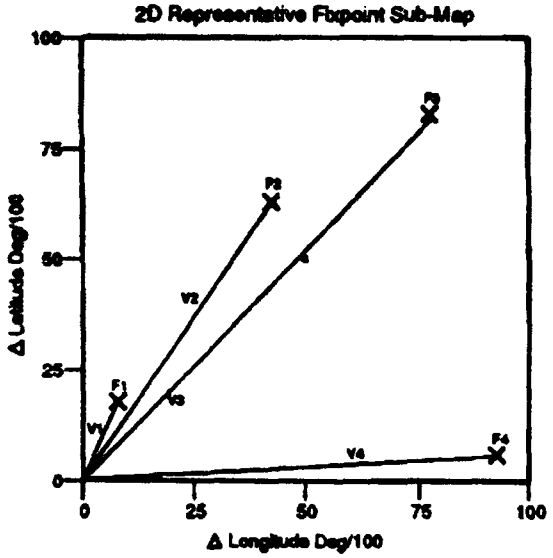


Figure 5-1. Conceptual Associative Memory Design

The associative memory is a data retrieval approach which conducts a parallel search of a large database returning a data record which is "closest" to an input record. The following example, see Figure 5-2, illustrates the application of this technique to the AFM fixpoint using two-dimensional fixpoint space with four fixpoints. The example encoding uses the difference in latitude, longitude, altitude, and a reference number to identify and differentiate fixpoints. The normalized matrix of fixpoints, S, is a n x 4 matrix where the rows represent the possible centroid of a multiple feature fixpoint region. Fixpoints include a fixpoint identifier, and a sensor type embedded in the type identifier.

Figure 5-3 illustrates the matrix based associative memory algorithm, and data forms used in the Fixpoint Catalog. The vectors, V, will contain the encoded fixpoints as previously illustrated. Vectors pertinent to the mission leg will be stored in the matrix, S, and their dot products will be used to derive the nonlinear function matrix, F. The diagonal values of the nonlinear function matrix are the maximum of the dot products of that row's fixpoint with the other row's fixpoints plus a heuristically-derived offset. The heuristically derived offset is a small constant added to the largest dot product to help distinguish a near miss with one row from a near miss with another row; this system uses a value of 0.05 (or five percent). Using the previously illustrated normalized storage matrix, S, from Figure 5-2, the



Encoded Fixpoints

	Rel.	Δ Long.	Δ Lat.	Δ Altitude
F1	1	7.5	17.5	53.50
F2	2	42.5	62.5	14.58
F3	3	77.5	82.5	42.67
F4	4	92.5	7.5	60.10

$$S = \begin{bmatrix} 0.02 & 0.13 & 0.31 & 0.94 \\ 0.03 & 0.55 & 0.81 & 0.19 \\ 0.02 & 0.64 & 0.68 & 0.35 \\ 0.04 & 0.84 & 0.07 & 0.54 \end{bmatrix}$$

S = Matrix of Normalized Fixpoint-Rows

Figure 5-2. Fixpoint Vector Encoding Example

unscaled non-linear function matrix is given by Equation 5-1, with the heuristic offset equal to zero.

$$F = \begin{matrix} 0.64 & 0 & 0 & 0 \\ 0 & 0.97 & 0 & 0 \\ 0 & 0 & 0.97 & 0 \\ 0 & 0 & 0 & 0.77 \end{matrix}$$

Equation 5-1

The final equation in Figure 5-3 is the retrieval approach. The input vector Vi is multiplied into the storage matrix resulting in a one dimensional vector of dot products between the approximate vector and the set of stored fixpoint vectors. The vector of dot products is multiplied by the non-linear function matrix resulting in a vector of scaled dot products. This vector is multiplied back into the storage matrix to extract the recall vector. For example, the 0.64 in the first row and column is derived from the fact that the first row of N has the highest dot product with row three; that dot product is :

0.64 = (0.02*0.02)+(0.13*0.64)+(0.31*0.68)+(0.94*0.35).

Any input vector having a dot product with a given row of S below the associated threshold value in the F matrix is more closely aligned with one of the other vectors in the storage matrix. Thus, if the dot product of the input fixpoint and a given row is below threshold value for the row, candidate fixpoint information associated with that row is suppressed.

The final equation in Figure 5-3 shows the method of retrieving a fixpoint vector from the storage matrix. This equation constitutes the parallel search in Figure 5-1. This approach can be applied iteratively, if necessary, to retrieve a single candidate fixpoint.

The associative memory described here was prototyped and tested with representative data; tests often yielded perfect recall. The algorithm, however, sometimes iterates to a deadlock solution, especially between fixpoints which have a small four-space separation. In these cases, either solution is equally

Example Algorithm

$V = (v1, \ldots vn)$ n Dimensional Vector

$$S = \begin{bmatrix} V1 \\ V2 \\ \cdot \\ \cdot \\ Vn \end{bmatrix}$$ n x n Storage Matrix

$$F = \begin{bmatrix} f1 & & & 0 \\ & f2 & & \\ & & \cdot & \\ 0 & & & fn \end{bmatrix}$$ Non-Linear Function Matrix

$V = S^T[F[S\,Vi]]$ Retrieval

Figure 5-3. Fixpoint Catalog Associative Memory Algorithm

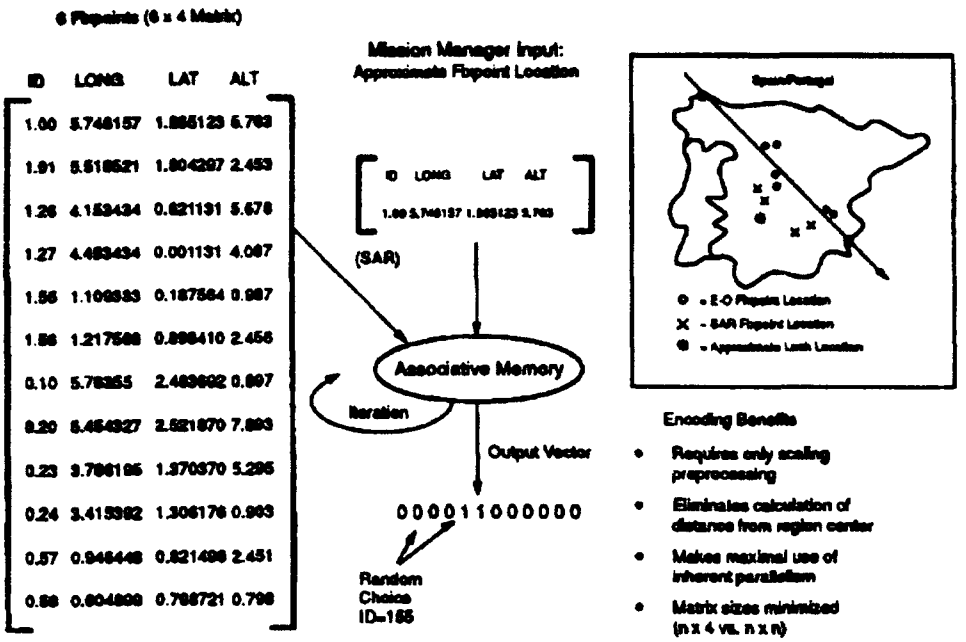


Figure 5-4. AFM Fixpoint Catalog Example

acceptable. A random choice between "tied" solutions appears to be a viable and reasonable design approach. This approach is illustrated in Figure 5-4.

6. FIXPOINT LOCATOR

The AFM Fixpoint Locator acquires and interprets sensor images to obtain navigation offsets which are input to the navigation filter. The Fixpoint Locator receives a fixpoint and an associated mission time to image it from the Mission Manager. The Fixpoint Locator sets up the sensor and compares a processed image to on-board databases, resulting in a navigation offset.

A block diagram of the AFM Fixpoint Locator is illustrated in Figure 6-1. The image interpretation is a multistep process involving traditional image preprocessing, neural network classification, and heuristic fixpoint location logic based on ATN interviews with strategic aircraft navigators. This logic returns a two-dimensional navigation offset which is passed to the Mission Manager through the AF communication interface. These system components are detailed in the following sections.

6.1 FIXPOINT LOCATOR AFO INTERFACE

The Fixpoint Locator Logic algorithm described above was encapsulated in an AFO which handles communication details among the Fixpoint Locator, Mission Manager, and Fixpoint Catalog. The AFO internal design also incorporates the translation capabilities for the navigation offset produced by the Fixpoint Locator. It is formatted into a navigation update message suitable for dispatch to the Environment/Navigation Simulation. The navigation update message contains the sensor dependent update type and the two dimensional navigation offset (in meters).

6.2 NEURAL NETWORK TECHNOLOGY FOR IMAGE PREPROCESSING

The Fixpoint Locator receives a fixpoint and a mission time to image the fixpoint from the Mission Manager. The Fixpoint Locator in an operational HVV would set up the sensor and capture the live image. In the simulation, a preprocessed image of the area of the fixpoint is retrieved from memory. The image interpretation is a multistep process involving traditional image preprocessing and neural network classification. The AFM development is not specifically concerned with the detailed sensor image acquisition and signal processing, only the result of

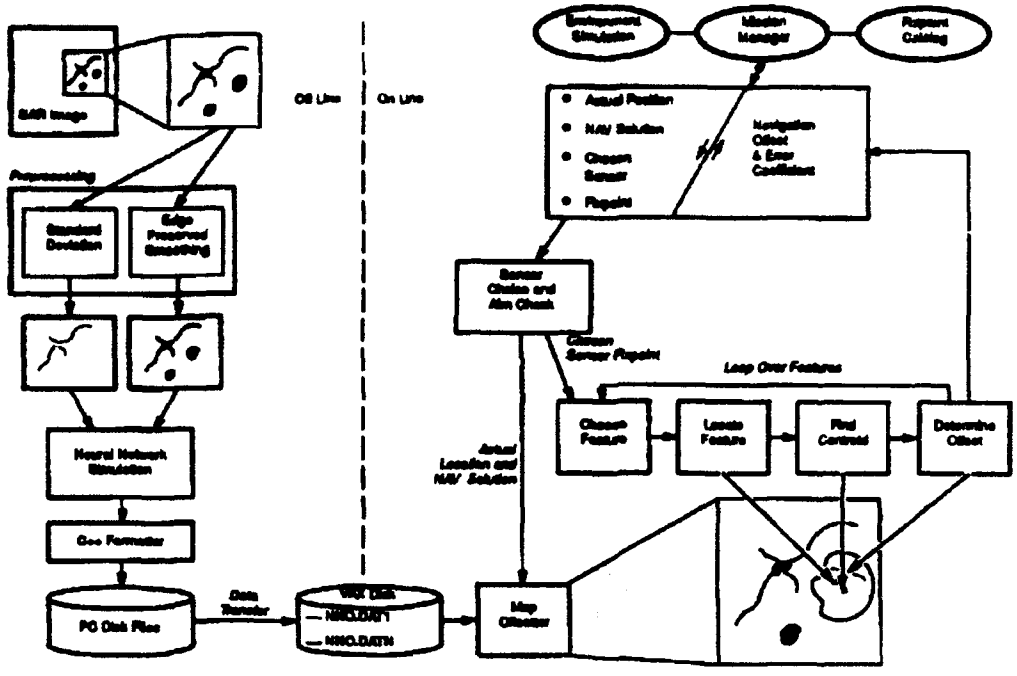


Figure 6-1. AFM Fixpoint Locator Overall Design

these processes on the errors incurred in the calculated navigation offset. As illustrated in Figure 6-1, the raw sensor image is first processed by the image processing techniques of window texture calculation (standard deviation) and edge preserved smoothing (EPS).

The standard deviation processing helps to remove false returns and dropouts while the EPS groups the feature pixels into closed regions. These processing techniques produce a binary output. The neural network takes this binary output and identifies features in the image. The neural network accomplishes this by performing a pixel-by-pixel determination of which image pixels are features of the image, and which are not.

The artificial neural network simulation model used is a three layer, fully interconnected back propagation system. It has two input nodes, eight hidden layer nodes, and two output nodes. The output of the neural network classifier is a dual floating point value which represents a binary number. For the purposes of the AFM simulation, the image pixel classification is done off-line. The SAR images used were unclassified SeaSat SAR images. If the resolution is assumed to be about 5 to 10 times better than actual, the images are representative of the HVV SAR returns. The dual-valued neural network outputs are reformatted into binary-valued-images which are used by the on-line portion of the Fixpoint Locator system.

The on-line portion of the Fixpoint Locator simulation reads the preprocessed sensor image and offsets it to represent the navigation and sensor pointing errors. The errors modeled are both systematic and random. The Fixpoint Locator simulation, therefore, requires that the true position of the HVV, which is maintained in the environment simulation, be passed to it. The preprocessed image is offset to represent the image coordinates which would have resulted from sensor imaging in an operational system. The fixpoint locator logic is then employed to locate the fixpoint features.

6.3 FIXPOINT LOCATOR HEURISTIC LOGIC DESIGN

The fixpoint location logic was developed based on interviews with FB-111 navigators conducted during the ATN system development. When searching a sensor image for a known fixpoint, the navigators first searched for large recognizable features, then smaller collateral features, until the location of the relatively small fixpoint could be reliably determined. This methodology has been adapted to AFM and tested using the preprocessed LandSat images. As

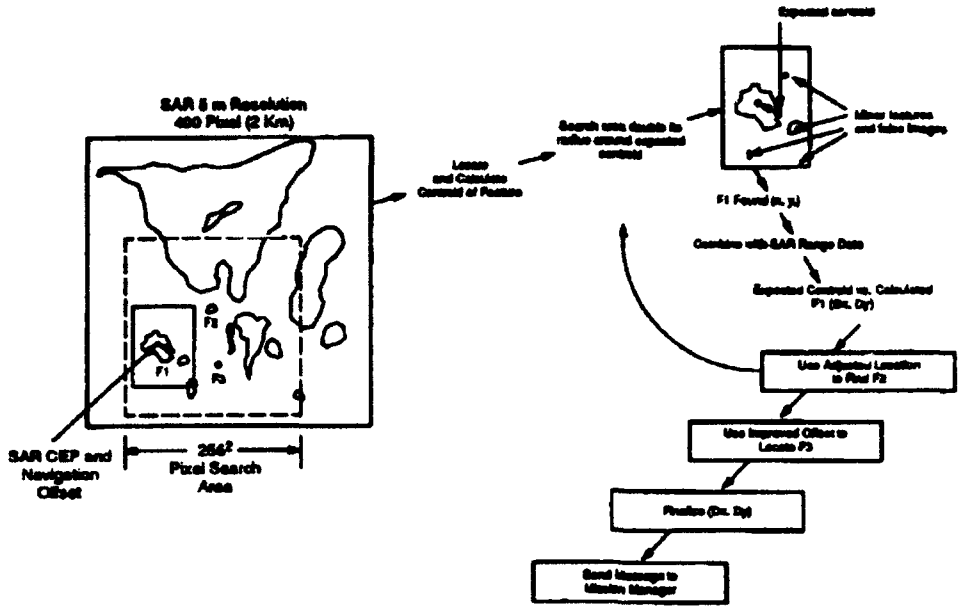


Figure 6-2. Fixpoint Locator Logic Example

illustrated in Figure 6-2, fixpoints are composed of a fine fixpoint, and two collateral features.

One of the collateral features has a radius approximately as large as the expected navigation error plus the sensor pointing error. This large feature must be in a region which is devoid of other features of the same approximate size and area. The Fixpoint Locator logic feature search begins at the point where it would be expected in the image. The search area is initially a heuristically-determined multiple of the feature radius. Note that by definition, this is a function of the expected maximum total offset. If the logic fails to locate the feature, this search space is expanded. Once the coarse feature is located, its centroid is calculated, and the offset between its expected position and the calculated position is used to locate the third, and final, fixpoint feature. The resulting offset is the navigation delta after being scaled out of pixel space into equivalent navigation measurements. Quantization error proportional to the sensor resolution exists, but with the fine resolution of the proposed SAR and EO sensors, (1-5 meters), the sensor boresight alignment errors dominate.

Within a feature search area, the Fixpoint Locator logic conducts a simple, connected pixel, region-building search. The feature is found and confirmed by the constraint that a fixpoint feature must be the largest localized feature within the approximate area expected. False identifications are minimized by this constraint as well as the relative geometry which must be established and confirmed with collateral features.

7. SYSTEM DEVELOPMENT AND INTEGRATION

The AFM system was designed and developed using an object-oriented programming based approach leveraging the real-time process/module integration benefits of Activation Frameworks. AF was implemented in C++ as a real-time sub-set of object-oriented programming techniques. The AF provides a consistent inter-agent interface, which utilizes object-based message passing and flexible focus of control using message importance and priority. The three top level system agents, the Mission Manager, Fixpoint Catalog, and Fixpoint Locator, were designed and developed independently once a set of messages and an associated prioritization scheme was established. Each of these agents were designed, from top level requirements and lessons learned in previous programs, by separate developers.

Conventional object-oriented programming paradigms, like C++, process a single thread of control through object messages which represent a sequence of data events. In a conventional C++ implementation the control of data event processing

follows processing paths of undetermined lengths. The AF is built on the C++ object model but at each object message initiation, a new processing thread is created or an old thread comes nearer to activation. Thus, each time an object relinquishes control of the computer process, system control is passed to an object which most urgently requires the processor or which produces output most important to the propagation of other process threads. The AFM AF uses this control to allocate computer resources to the agent and agent sub-objects.

The design of each agent was selected to optimize required aspects of agent performance. The Mission Manager was designed as a set of C++ objects which were encapsulated in the AF as AFOs. Because of the success of the ATN mission planning system, the Mission Manager was designed as a large number of AFOs, which separated the AI and conventional programming modules into a set of objects. These objects were tightly integrated using a fixed set of messages to enhance real-time control. The Fixpoint Catalog was designed as several C++ objects and some procedural code which was encapsulated as a single AFO to enhance data retrieval speed and control data passing and translation activity. The Fixpoint Locator was designed as a C++ object with a large procedural content. The Fixpoint Locator works with large images performing several stages of processing on the input image. This image processing approach included some conventional processing algorithms which were procedural in nature, calling for the localization of the object.

The Mission Manager developed under the ATN system was originally developed around an expert system and causal network in the LISP language. Real-time constraints forced the entire ATN system to be written in C. The ATN Mission Manager operated as a subset of the system AFOs, with a gateway AFO used as the sole input/output port to other top level agents. The ATN gateway AFO re-sends received input messages to Mission Manager AFOs. This gateway AFO and the loose coupling of message data structures, AFO data structures, and the AFO scheduling system made modification and upgrade of the system difficult. The requirement of a gateway AFO and the difficulty of specifying, on a system-wide basis, the method for decoding particular messages spawned the requirement to move toward an object-oriented AF system for the AFM program. The ATN gateway AFO illustrates that a single level of object separation, scheduling, control, and aggregation is also not sufficient in a complex application like navigation mission management. The AFM Mission Manager uses the list oriented mission decomposition and planning system implemented under a C++ based AFO. The AFM causal network engine also takes advantage of object-oriented encapsulation under a list processing system shared with the other planning AFOs.

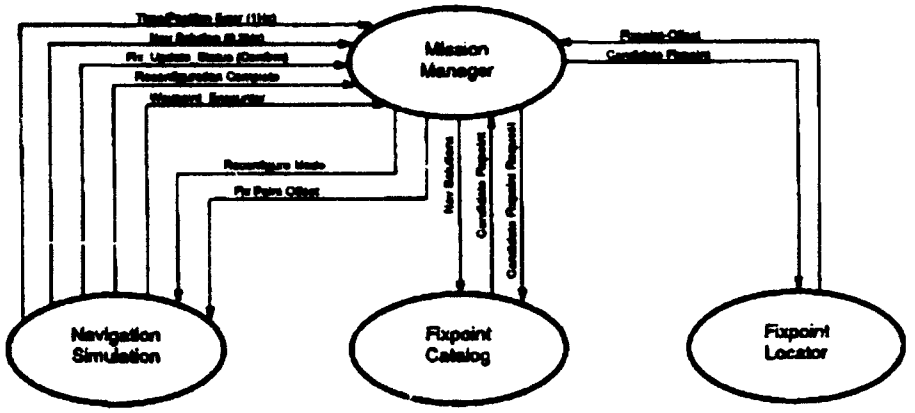
The AF system exploits the object-oriented benefits of C++ in several critical ways. AFOs are used to encapsulate both procedural code and objects which are called or accessed by an incoming message. The AFOs are complex enough to know if sufficient data has arrived for sub-object or procedure activation. The messages in the system are also objects, each having a consistent extraction mechanism as well as a consistent scheduling interface. The messages use object methods and overloaded operators to facilitate these features.

The list processing system is a C-based system which pre-dates the ATN program. The object-oriented features of C++ including inheritance has been used to encapsulate this system and make it available to multiple other system objects. The planner object uses the list processing system extensively. The encapsulation of existing routines was used several places in the AFM system.

The messages illustrated in Figure 7-1 comprise the integration design of the complex agent models used in the AFM system. Top level agent decomposition was determined according to avionics equipment, function, and existing models (i.e., the mission manager and navigation simulation). The small number of messages indicate that the system decomposition was successful at maximizing agent encapsulation while minimizing the passing of data. The most difficult integration challenge was the integration of the PC-based Navigation Simulation, a stand-alone simulation which was developed over several programs, into the AFM simulation. The illustrated messages between the Navigation Simulation and the Mission Manager agent occurred over a RS-232 serial line. The serial link necessitated the use of the handshaking signals, Fix_Update_Status and Reconfiguration_Complete. Except for the necessary handshaking messages, the design represents an exceptionally clean design.

8. CONCLUSIONS

AFM was a challenging application which required seamless real-time integration of several advanced technologies to demonstrate the feasibility of an autonomous navigation fixtaking capability. The AFM system simulation components were developed through prototyping and verified with representative databases to ensure real-time performance. In order to organize the AFM program into a manageable system, a hierarchical decomposition of the mission planning and data interpretation functions were utilized. Within the hierarchical organization, an object-oriented design and integration approach facilitated



- Navigation simulation provides Navigation solutions and reconfigures mode
- Fixpoint catalog returns candidate fixpoints from Point of interest vector
- Fixpoint locator returns deltas from candidate Fixpoint vectors

Figure 7-1. Mission Manager Intermodule Interface Input

agent paradigm encapsulation, provided consistent inter-agent communication, and enabled real-time control prioritization. The AFM software modularity and flexibility were further enhanced by taking advantage of the object-oriented capabilities of C++ in implementing the AF. With the use of the AF architecture, individual software modules at the agent level were designed to communicate using the same message passing paradigm.

In order to manage the system, locate fixpoints, and retrieve fixpoints from databases three major object agents were developed. These agents required diverse computational approaches to meet the real-time performance requirements of the HVV environment. The AFM Mission Manager was developed based on extensive lessons learned from the ATN design. This Mission Manager utilized the AF to embed intelligent methodologies on mission planning, analysis, and moding. The Fixpoint Catalog demonstrated the real-time capable fixpoint retrieval techniques based on advanced associative memory technology. The Fixpoint Locator simulation achieves efficient image feature recognition by emulating human feature identification processes captured through interviewing tactical aircraft navigators. The image processing techniques achieve automated fixpoint location and promise real-time operational performance through the application of advanced parallel processing algorithms. The module design benefited from the algorithm encapsulation which allowed independent development of agent modules. A well defined real-time control approach with consistent inter-module communication interfaces allowed simultaneous module and agent development by multiple developers.

The results and experiences gained from this program not only demonstrated the feasibility of automating the navigation fixtaking task but, equally important, demonstrated the integration of several diverse technologies working in concert. Through the use of software techniques such as hierarchical organization, object-oriented design, and the AF data fusion methodology real-time performance was achieved.

REFERENCES

1. Hancock, J.P.,Thomas S., Development and Real-Time Laboratory Demonstration of an Autonomous Fixtaking Management System, TASC, WL-TR-92-1031 (AD-B167-988), May 1992.

2. Glasson, D.P., et al., Development and Real-Time Laboratory Demonstration of An Adaptive Tactical Navigation System, TASC, WRDC-TR-90-1034 (AD-B151-995), May 1990.

3. Green, P.E., AF: A framework for real-time distributed problem solving, Collected papers of the 1985 distributed AI workshop, Sea Ranch, CA, pp. 337-356, November 1985.

Discussion

Question K. ROSE

What navigation accuracies can be achieved with the AFM system?

Reply

This system was designed and tested through simulation for approximately 60 meters accuracy.

THE DEVELOPMENT PROCEDURES AND TOOLS APPLIED FOR THE ATTITUDE CONTROL SOFTWARE OF THE ITALIAN SATELLITE SAX

G.J. Hameetman and G.J. Dekker
National Aerospace Laboratory NLR
P.O. Box 90502, 1006 BM Amsterdam
The Netherlands

1. SUMMARY

The Italian satellite (with a large Dutch contribution) SAX is a scientific satellite which has the mission to study röntgen sources. One main requirement for the Attitude and Orbit Control Subsystem (AOCS) is to achieve and maintain a stable pointing accuracy with a limit cycle of less than 90 arcsec during pointings of maximal 28 hours. The main SAX instrument, the Narrow Field Instrument, is highly sensitive to (indirect) radiation coming from the Sun. This sensitivity leads to another main requirement that under no circumstances the safe attitude domain may be left.

The on-board software that controls the SAX AOCS must therefore be highly reliable with respect to the safeguarding of the satellite attitude. On the other hand, the scientific character of the mission imposes flexibility requirements to the software, as during the mission the need for new (or changed) observation types may arise. These have to be implemented by loading updated software during the mission.

The AOCS on-board application software is being developed by NLR using a suite of CASE tools consisting of the Teamwork package (prescribed for all SAX on-board software), the CADESE software configuration management package, processor emulation hard- and software and a number of special test and simulation tools. The AOCS processor and its operating system have been developed in parallel by another firm.

The paper describes the application software in relation with the overall SAX AOCS subsystem, the CASE tools that have been used during the development, some advantages and disadvantages of the use of these tools, the measures taken to meet the more or less conflicting requirements of reliability and flexibility and the lessons learnt during development.

The quality of the approach to the development has been proven the (separately executed) hardware/software integration tests. During these tests a neglectible number of software errors has been detected in the application software

2. INTRODUCTION

In the early 1980's the Italian Space Agency (ASI), together with the Netherlands Organisation for Space Research (SRON), started the development of the SAX satellite project (Ref. 1). Mission of the SAX satellite is to perform a systematic and comprehensive observation of celestial X-ray sources in the 0.1-200 keV energy range with particular emphasis on spectral and timing measurements. The mission is funded by the Italian and Dutch national space agencies, resp. ASI and NIVR. The satellite (Fig. 1) will be injected into a circular equatorial orbit with an inclination of less than 5° and an initial altitude of 600 km by an Atlas Centaur launcher in the last quarter of 1994. The nominal mission lifetime is two years, with a design goal of four years.

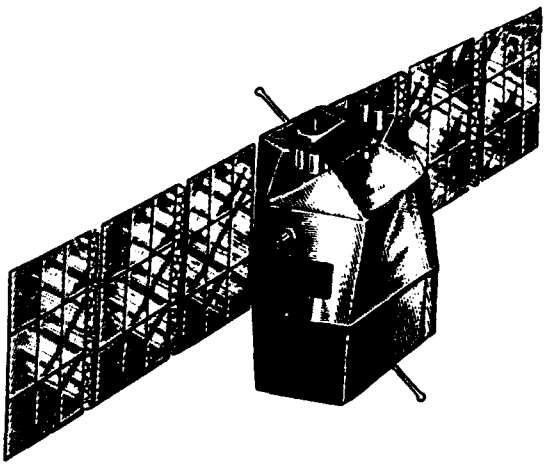


Fig. 1 SAX satellite

The Attitude and Orbit Control Subsystem (AOCS) of the satellite is under development by Fokker Space and Systems (FSS) under contract with the satellite prime contractor Alenia Spazio (Roma, Italy). As subcontractor of FSS, the National Aerospace Laboratory NLR develops the Application Software for the AOCS on-board computer (Ref. 2).

In this paper, a short overview of the application software in relation with the overall AOCS subsystem is given. The main requirements on the software are discussed. Besides requirements on a high pointing accuracy, it is important that the software is highly reliable and yet flexible for future (in-orbit) updates. The design measures to obey to these more or less conflicting requirements are described.
During the development of the application software, a suite of Computer Aided Software Engineering (CASE) tools is applied. Our current experience with these tools, in terms of recognised advantages

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

and disadvantages is given. This experience is related with previous projects that were similar in size and complexity, such as the on-board software for the satellites ANS and IRAS and the software for groundstations.
Most of the software has been integrated with the AOCS computer and tested in a simulated environment. During these tests only a few errors have been found, illustrating the quality of the development approach. The paper ends with some lessons learnt during the development.

3. THE SAX AOCS APPLICATION SOFTWARE

The AOCS subsystem hardware consists of (Fig. 2):
- The Attitude Control Computer (ACC). This computer is based on a 80C86 microprocessor extended with a 8087 co-processor. The ACC is fully redundant. Two identical, independent, computers are integrated into one unit. One of the two is cold standby.
- A set of attitude sensors. These are:
  * Three Sun Acquisition Sensors (SAS).
  * Two Quadrant Sun Sensors (QSS).
  * Two Magnetometers (MGM).
  * Four gyro's (GYR).
  * Three Star Trackers (STR).
- A set of actuators. These are:
  * Four Reaction Wheels (RWL).
  * Three Magnetic Torquer Rods (MTR).
  * A Reaction Control Subsystem (RCS) containing thrusters that will be used for variation of the satellite's velocity in orbit.
- A set of service units:
  * A Monitoring and Reconfiguration Unit (MRU).
  * A Power Distribution Unit (PDU).

These units are connected to each other by a 'Modular Attitude Control System bus' (MACS-bus). The AOCS subsystem is connected to the other units if the satellite via an 'On-Board Data Handling bus' (OBDH-bus).

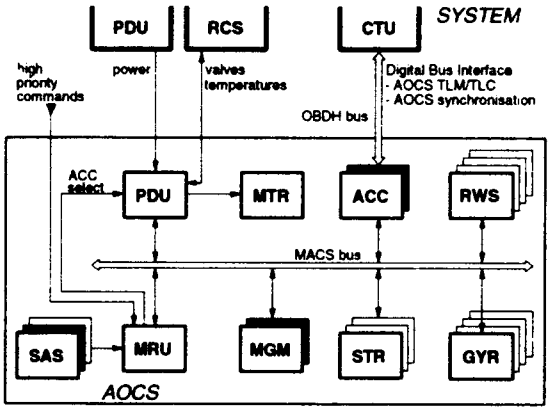


Fig. 2 Simplified block diagram of the SAX AOCS

The AOCS is controlled by software running in the ACC. This software is divided into two packages:
- Basic Software (BSW), developed by Alenia Spazio that provides operating system services and basic interface services to the MACS and OBDH busses. It hides the ACC hardware peculiarities for the Application Software.
- Application Software (ASW), developed by NLR. This software provides all attitude control tasks and manages the application-dependent communication with the ground (via the OBDH) and with the AOCS units.

The basic requirements on the AOCS and hence on the ASW software are to provide the capability to:
- Command the Zc-axis (main instrument axis) of SAX to any direction (within the pointing constraints) with an accuracy of 90" and the Yc-axis with an accuracy of 16.5 arcmin. The commanded attitude must be maintained during pointing periods of maximal 28 hours.
- Safeguard the satellite against violation of the safe pointing domain. This safe pointing domain is defined mainly by the fact that the main scientific instrument, the Narrow Field Instrument (NFI), will be destroyed by (indirect) radiation from the sun. Therefore, the angle between the Zc-axis of the satellite and the sunvector must be at least 60°.
- Autonomously acquire and maintain a safe attitude when no ground commanding is available and after safeguard violations.
- Process ground commands and generate relevant telemetry for health checking on the ground and for attitude reconstruction in relation with the processing of the acquired scientific data.

Due to the high sensitivity of the main scientific instrument to sun radiation, the correct execution of the ASW is critical for the success of the mission. Therefore, it has to be highly reliable and measures have to be implemented to ensure the survival of the satellite under all foreseeable single point failure conditions. On the other hand, previous experiences have learnt that such software shall also be flexible and adaptable to new requirements and/or unexpected situations during the mission. Due to this type of flexibility the missions ANS (also an X-ray mission) and IRAS (Infrared research) were highly successful. Without this flexibility, both missions would have been largely or completely failed (Refs. 3 and 4). In scientific missions like these, the scientific data obtained can lead to requests for new observation types or for changes in existing observation types. It can also appear that the sensors and/or actuators used show unexpected behaviour, which has to be compensated for in the AOCS software. Therefore the need exists that updated versions of the software can be uploaded (via the telecommand channel) into the ACC during the mission.

4. DESIGN MEASURES TO MEET THE REQUIREMENTS

In a combined hardware/software system, both hardware and software failures have to be taken into account for a reliable design. The result of hardware failures can be assessed via a traditional Failure Mode Effect Analysis (FMEA).
NLR research to take also software failures into account in a FMEA analysis (Ref. 5) has learnt that random software failures (such as random addressing the memory) are difficult to handle in this respect as the effect of such failures cannot be predicted. One conclusion of this study was that the design of the hardware/software system has to be such that the safety critical software is protected against such random failures. For SAX, the following design measures have been taken:
* The ASW software has been divided into two parts:
  - Basic Attitude Control (BAC) software. This part is highly reliable and safe. It will be (initially) stored in Read Only Memory (ROM) in the ACC, together with the BSW. Main purpose of this software is to provide the functionality needed to acquire and keep a safe satellite attitude after power-up and fallback. Furthermore it contains the data handling functions needed to submit the state of the AOCS to the ground and to give the control over (on ground command) to the EAC

software. After a fallback to the BAC software, the ground operations team can analyse the reason for fallback and devise solutions to circumvent this reason.

- Extended Attitude Control (EAC) software. This software provides all functions required for the AOCS, including the control laws for accurate scientific pointings and the generation of full attitude reconstruction telemetry. EAC is stored in Random Access Memory (RAM) of the ACC and can be loaded and/or modified on ground command.

* The 8086 facilities to separate code from data are used extensively. These facilities provide a hardware protection against overwriting the code segment by the application software.
* The behaviour of the software is checked by a hardware ACC watchdog monitor. This monitor has to be triggered at least once per two seconds with a correct bit pattern. If it receives a wrong bit pattern, or if it is not triggered at all, it forces an ACC power-up initialisation sequence and hence a fall back to the BAC software. This watchdog monitor protects the system against failures like endless loops or a complete halt of the software.
* A similar watchdog is implemented outside the ACC. This watchdog is located in the Monitoring and Reconfiguration Unit (MRU). The ASW regularly has to read a checkword from this unit via the MACS bus, change one bit in this checkword, and send it back to the MRU. If the MRU detects that no checkword has been received back for six seconds, it will force a switch over to the cold standby ACC unit. This monitor protects against failures in the ACC hardware and within the MACS bus hardware. It also prevents that repeated initialisation of the active ACC by the ACC watchdog leads to loss of attitude control.
* For reasons of power consumption, the BAC software cannot be executed from ROM directly. Therefore, it is copied to RAM as part of the ACC power-up initialisation sequence. In order to check that the copy of the software is not corrupted during execution, a checksum check is executed twice a second on the copied code. If this check fails, the ACC is forced to execute a power-up initialisation sequence, by a false trigger of the ACC watch dog.
* During the execution of the EAC software, a separately coded safeguard function is executed every half second. This safeguard function checks that the behaviour of the satellite remains within the safety limits. If the attitude becomes unsafe, the ACC is reinitialised via the ACC watch dog, leading to a fallback to the BAC software. The safeguard software is executed in the context of a special interrupt. In this way, the safeguard code will be executed regardless of the current state of the EAC software. The safeguard criteria have been designed as simple and robust as possible in order to prevent errors in the safeguard code.

The reliability of the ASW has been designed-in by the strict application of interface control mechanisms and configuration control measures. Interface control has been executed during the design by application of the Teamwork package (refer to section 4). For the coding phase, we have copied the Ada 'package' mechanism to the ASW implementation language C. Every module in the ASW has been declared by a header file (that defines the module prototype) and defined in a code file (that contains the actual executable code of the module). Similarly, every shared data item has been declared in one header file and defined in one code file. A module that calls another one must

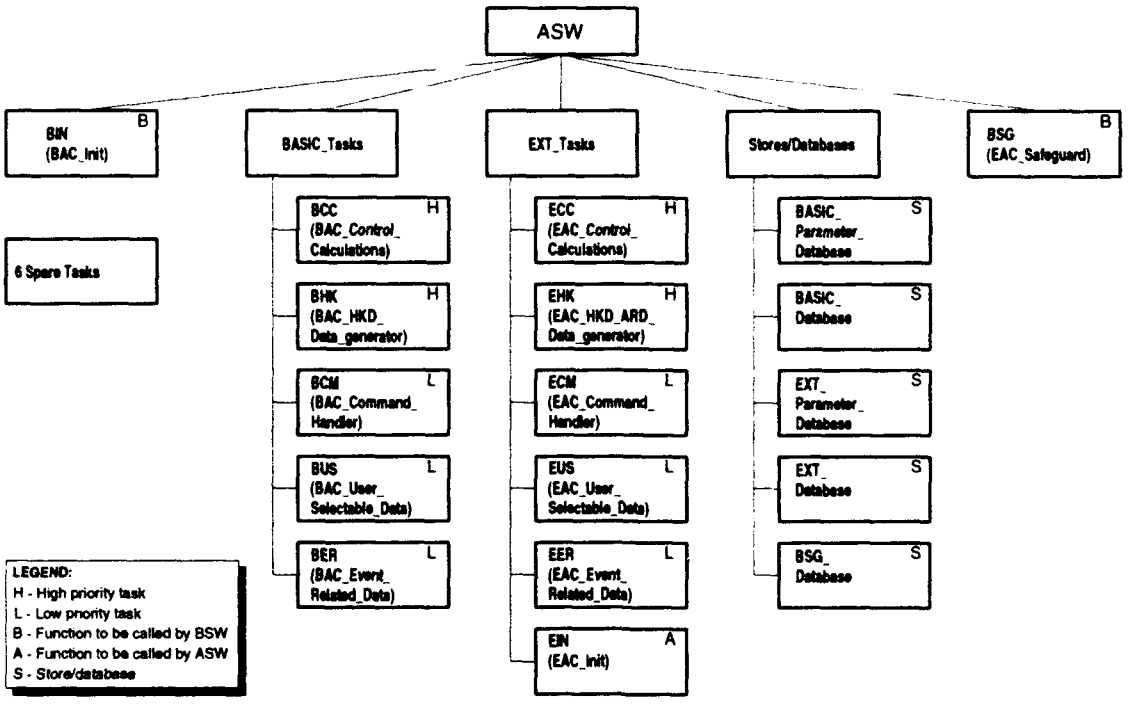


Fig. 3 ASW components

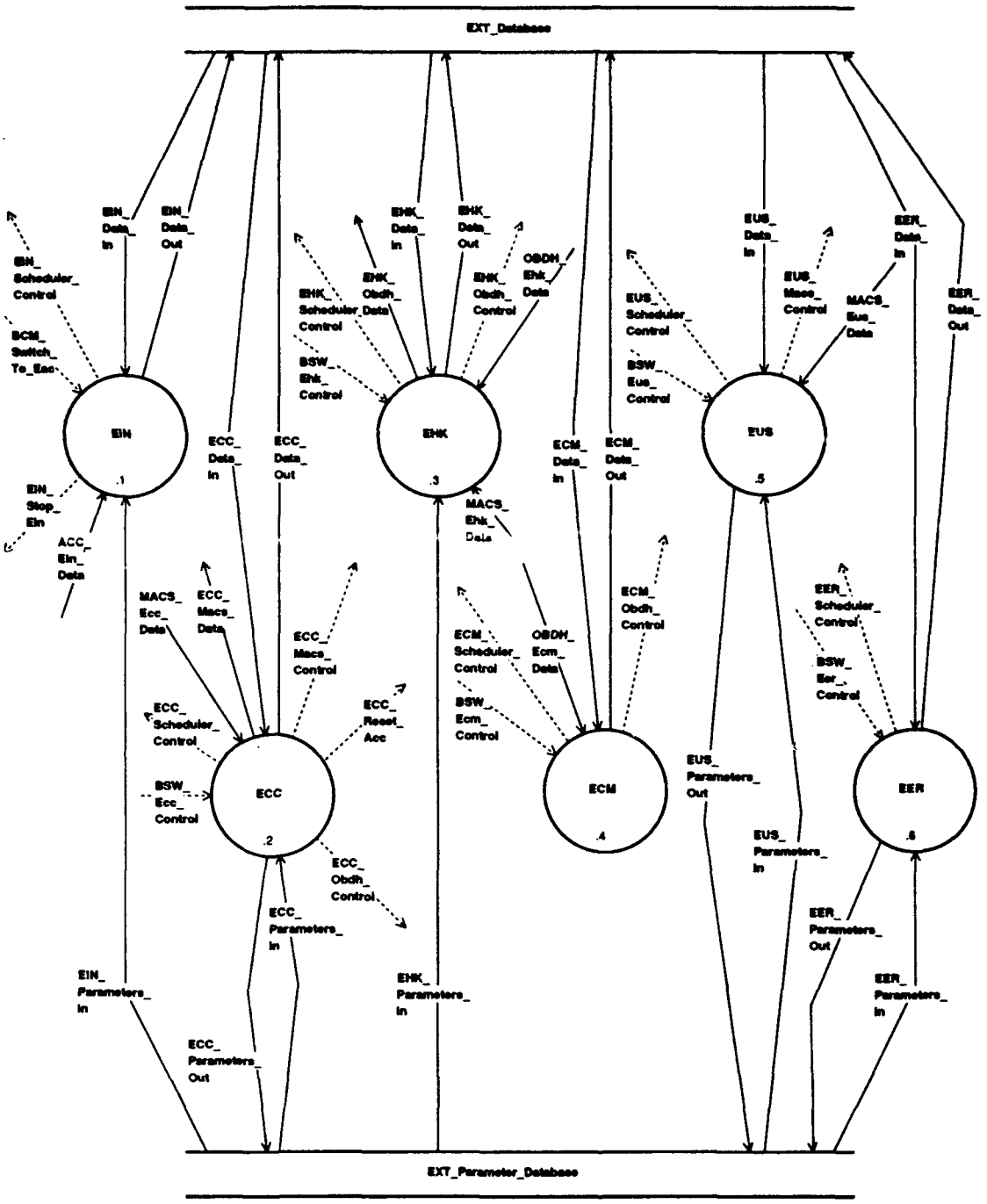


Fig. 4 Basic architecture of the EAC software

'#include' the relevant header file, which ensures that the compiler can check the correctness of the calls. Similarly, the declaration of shared variables has to be included in all modules that use them. The header files are derived directly from the Teamwork database, thus ensuring the compatibility between the detailed design and the code. All files are under control of the configuration management package CADESE (see section 4).

The performance of the design and control laws to be implemented by the ASW has been checked by FSS during extensive simulations of the system.

The required flexibility is achieved by designing the EAC software as a separate independent package, that can in principle execute independent of the BAC software. The functionality of the EAC software has a significant overlap with the BAC functionality in the areas of command processing,

telemetry generation and unit data handling. But in order to be free to change software related to these aspects, all EAC functions have been designed as separate programs, that are executed independent of the BAC programs. Figure 3 contains an overall overview of the ASW. Figure 4 shows the structure of the EAC software,which is similar in construction to the BAC software. Both packages implement the full set of functions, which are executed cyclically twice per second:

- Initialisation (performed only once);
- Control calculations (including unit handling and actuator commanding);
- Telemetry generation (House Keeping Data and Attitude Reconstruction Data);
- Telecommand processing;
- User selectable data processing; and
- Event related data monitoring.

The independence of the EAC software means that for instance the telemetry processing, which is nearly identical for BAC and EAC is designed as a separate EAC task. In the current design, this EAC task makes extensive use of the routines in the corresponding BAC task in order to save memory occupation. But it can be replaced completely by software that is independent of the BAC software and that implements different requirements, if that would be necessary during the mission. This would be much more difficult if the BAC task for telemetry generation was used completely during the execution of the EAC software.

It should be noted that a high flexibility of the software can lead to a decrease of the reliability. During the mission, the temptation can be high to load updated software that has a lower qualification status than the original EAC software. The design measures to protect the BAC software against random EAC software failures, however, guarantee the survivability of the mission, even if this software is replaced by lower quality software.

5. CASE TOOLS IN USE

For the development of the ASW, a development environment has been purchased, based on a network of four HP 64000 workstations. The following development tools have been installed on this network:

- The CADRE Teamwork package, that has been prescribed by the prime contractor for all parties developing on-board software for SAX. Teamwork has been used to input and check the Software Requirements and the Architectural Design. It supports the Structure Analysis method for (real-time) software systems as defined by Yourdon, DeMarco, Constantine and Hatley. The requirements and architectural design have mainly been described in Data Flow Diagrams and State Transition Diagrams. The data that is handled by the ASW has been described in detail in a Data Dictionary. Teamwork includes a powerful checking option to ensure the internal consistency of the thus obtained functional decomposition.
  The Detailed Design of the ASW has been developed using the Structured Design method, also supported by Teamwork. Each individual task of the ASW is defined with one Structure chart and a number of Module Specifications. Also for this development phase, Teamwork provides a number of consistency checks, although these are less powerful than the checking options for the Structured Analysis method.
- The DeskTop Publishing package Framemaker. Framemaker is an integrated text and graphics processing tool with the capability to import the graphical information from Teamwork models. Using this tool, the text and pictures of the design can be quickly combined into complete documents via the definition of generic document, refering to text and Teamwork models.
- CADESE. The tool CADESE (purchased via the Dutch firm ACE) is an extension of the standard UNIX SCCS facilities. It is used for the configuration management of the developed code. CADESE offers version control to maintain the source files for each official software release and to ensure that earlier versions can easily be retrieved. Furthermore, it is able to generate UNIX 'make' files for each release, such that the binary code can be generated automatically after each change.

In addition, a number of testtools have been purchased, or have been developed especially for the ASW development. These testtools are not considered as real CASE tools, but they are of the same level of importance for the development of the software.

- 8086/8087 Hardware emulator. This emulator is in fact a real 8086/8087 processor, enhanced with options to control the memory contents and the execution from within the HP development system.
- Software emulator of the 8086/8087 processor on the HP 64000 development system. This emulator was used mainly for the individual module tests, after module tests executed in the native C environment on the HP system. The use of the emulator ensures that the modules tested in the HP native environment will also execute correctly in the target processor. Also, this software emulator decreased the work load on the hardware emulator.
- The ACC-Command-Generator is a program running on the HP development system and interacting with the user. It generates ACC telecommands in human-readable/editable form in user specified ACC-Command-Files. Via a menu-based interface the user can select new ACC-Commands to be generated, and specify their parameters. The program can run in checking mode and in non-checking mode. In the checking mode all ACC-Command parameters are enforced to be valid. In the non-checking mode these enforcements are left out, so that illegal ACC-Commands can be generated in the ACC-Command-Files.
- The TMTC-Simulator is a program running on the HP development system and interacting with the BSW-Simulator (which simulates the BSW behaviour). From the standard input stream it reads ACC Commands from a redirected ACC-Command-File as generated by the ACC-Command-Generator, and passes these to the BSW-Simulator. It reads telemetry buffers passed by the BSW-Simulator, and displays these in human-readable form. The program also can run in checking mode and in non-checking mode.
- SATSIM is a program running on the HP development system and interacting with the BSW-Simulator. It simulates the SAX sensors, actuators and dynamics. Using this program, closed loop integration tests have been executed.
- The BSW-Simulator is a program running on the Intel 8086 emulator system and interacting with the user, the ASW, the TMTC-Simulator and SATSIM. It simulates the Basic Software. The BSW-Simulator was needed because the BSW is developed in parallel with the ASW. The simulator is a very simplified version of the BSW with the emphasis on the properly modelling of the ASW/BSW interface.

6. ADVANTAGES AND DISADVANTAGES OF USING CASE TOOLS

During the design of the ASW, a number of advantages and disadvantages of the tools used have been recognised (in comparison with the manual application of the structured methods).

A main advantage of tools like Teamwork is the existence of automated checks that can be done on the design. These checks guarantee the internal consistency of the design models (functions and dataflows) of the software. Decomposition of a function with its dataflows into a set of lower level subfunctions can be done without introducing inconsistencies between the various design levels. This point is especially important during the (iterative) initial design of a model. The consistency checks enable a more detailed decomposition of the model. Our experience is that we could add two or three levels to the design hierarchy, without losing control over the design. In this way, more design details could be formalised (and checked) than in previous projects, leading to an increased quality of the design.
It should be noted, however, that the checks that are possible for Structured Analysis models are more powerful than those for Structured Design models. This is mainly due to the fact that SD models are not hierarchically leveled.

The guarantee of internal consistency also enables a more safe implementation of design changes. The addition or deletion of a dataflow on a low level of decomposition has to be reflected also in the higher levels where appropriate. Tools like Teamwork do not allow that loose ends are left in the design, like data that is used, but never generated. The manual application of the used methods imposes risks in this area. In our situation, FSS developed the detailed functional design, while NLR was developing with the detailed software design (Ref. 6). This approach was necessary in order to optimise the development schedule, but it would have been impossible without the Teamwork tool, due to the iterative nature of this approach.

The combination of Teamwork with a desktop publishing tool has led to a fast document production process. The contents of a document, for instance the Architectural Design Document, has been defined in terms of text portions and the Teamwork model identification of the design model. Using a specialised interface tool between Teamwork and Framemaker, the text and graphics of the documents can be automatically combined into one printed document. This integrated approach enabled us to produce a complete issue of design documents within one working day. In comparable previous projects, such production could take one week or more and led to lower quality documents.

A final advantage of the use of a computer aided tool to support the design method is that changes to the design can be (and in the Teamwork approach is) logged and identified. Every issue of a Data Flow Diagram can be saved and is accompanied by a status record. This information can be of help to determine the cause of unexpected errors. Old issues can be used as backup information and in cause of unexpected problems the cause can be traced back.

A similar advantage can be mentioned for the use of the CADESE configuration management package. CADESE, together with the well-known SCCS software, logs all changes to coded modules and includes facilities to roll back to earlier versions. CADESE includes an utility to generate a proper 'make-file' for the software, that includes all file dependencies. This utility guarantees that the proper version of all code files is used to produce binary load images of the code. Interface problems that can easily occur by using invalid (outdated) precompiled object files are thus eliminated, without raising the burden to compile all sources each time a new load image is needed.

A disadvantage that we encountered was that the application of a tool like Teamwork is likely to become a design goal. In a project where the methods are applied manually, it is natural to deviate from the formal definition of the methods in those cases where such formalism would unnecessarily restrict the designers. When a tool is used to check the obedience to formal rules, such deviations are not allowed the designers are tempted to define tricky work arounds, just to satisfy the method checker. It appeared difficult to regard the tool just as a tool and not as a design goal.
Similarly, the availability of powerful graphical editing facilities on the diagrams tempts the designers to spend (lose) time in the esthetical optimisation of these diagrams. Such optimisation does not improve the design, nor does it increase the quality of the documentation. It is difficult to recognise these temptations and to counteract them during the development.

7. PRACTICAL PROBLEMS ENCOUNTERED WITH THE APPLICATION OF CASE TOOLS

During the development of the ASW software, a number of practical problems have been encountered with the application of the used tools. These problems are reported such that tool developers can enhance their products. Unless explicitly stated, it is our opinion that the reported problems are not specifically related to the Teamwork tool, but have a more general nature.

The major problem encountered is the absence of a coupling between the Structured Analysis and the Structured Design method. These methods cannot be used in an integrated way in one model, such that automated checks of the detailed design (modelled using the SD method) against the architecture (modelled using the SA method) can be executed. The absence of such checks counteracts the advantages of using these methods in relation with an automated tool.
From a software developers' point of view, it should be possible to define a low-level (SA) process in the form of a (SD) structure chart with related module specifications.

Another problem was the rigor with which the configuration management of models has been implemented. Our basic approach in developing the ASW architecture has been to define the task structure as 'baselined' model. (A baselined model cannot be changed, unless its owner removes the 'baseline' status.) Thereafter, the individual developers detail one or more processes of this model in a private 'incremental' model of the baseline. This approach allows multiple developers to work independent of each other in parallel. However, after definition of incremental models to a baselined model, the 'baseline' status cannot be removed, so there is no way at all to change the latter model. This has as result that errors or omissions cannot be corrected in the baseline, but have to be corrected in all separate incremental models individually.
Furthermore, the incremental models cannot be

integrated in the baseline model itself, but have to be integrated into another incremental model of the baseline, which can then be the baseline for further development. This results in an artificial hierarchy of dependant models that is not suited to explain the current design status after a number of iterations.
Our (pragmatic) approach has been to make baselined models 'complete' (which means that all dependencies to the parent model are removed by copying the information of that model into the current model) and link completed models to a dummy (empty) model. In this way several issues of a baselined model can be controlled at the same level of model hierarchy.
The rigorous configuration management has also led to the decision that the architectural design of the ASW would be documented in another model than the detailed design. Both models own their private data dictionaries that define the same dataflows. Only rigid, but manual, configuration control procedures can ensure the compatibility of these data dictionaries.

A practical difficulty with Teamwork (but probably with more tools) has been that process specifications and module specifications have to be defined in a textual form. It is not possible to define a specification in a graphical way, for instance in the form of a Nassi-Shneiderman chart (Ref. 7). We have used a very simple form of pseudo code for the module specifications to compensate for the absence of graphical representation of detailed module logic.
Furthermore, the text editing capabilities of the tool are rudimentary and limited to individual model entries (data definitions, process specifications, etc.). No options exist for instance to change a certain term in all P-specs or M-specs with one command.
Improvements in these areas will prove very helpful for software developers.

8. CURRENT PROJECT STATUS

All ASW software has been coded and module tested. The BAC software, consisting of ± 9000 lines of executable (non-comment) C-code, has been integration tested by the development team in the development environment. During these integration tests, the software requirements were verified. After that, the software has been subjected to a number of validation tests executed by an independent test team. The validation tests involved closed-loop real-time tests of the software running in a "FUnctional MOdel" (FUMO) of the ACC computer. The ACC environment, including the satellite dynamics and the unit behaviour, was simulated as realistic as possible. During these tests, all relevant User Requirements for the BAC software were checked. During the validation tests a total of 5 errors have been identified in the BAC code under test. The majority of these errors has been traced to late requirement changes, which were not implemented completely, or led to unexpected side effects.
After the validation tests, the BAC software has been formally delivered to Alenia for used in the AOCS subsystem integration process, where the ACC is integrated with the real units. During these tests, no errors in the code have been identified yet.

The EAC software, consisting of ± 5000 additional lines of executable C-code is being integration tested by the development team. The independent validation tests of this software has been started.

9. LESSONS LEARNT

During application of the described CASE tools, a number of lessons have been learnt that seem to be valuable for future developments.

- The application of an automated tool to support the Structured Analysis and Structured Design methods during the design increases the quality of the software design and thus contributes to the overall success of the mission. However, does not lead to a decrease in design cost.
- In case a larger Structured Analysis model is developed by a number of team members in parallel, it is important to define a number of standards to be adhered to. Such standards should at least cover topics like naming conventions for processes and dataflows; use and documentation of control signals (triggers and/or levels); pseudo code to be used inside process specifications and the overall layout of the DFD's.
- During the development of a Structured Analysis model of a complex piece of realtime software, the design of the dataflows is as important as the design of the processes. Ignorance of this can lead to the situation that various parts of the model use effectively the same dataflows, but define them from different viewpoints. This can increase the model-integration effort.
- In a multinational project where a number of firms cooperate to develop a large integrated software system, the application of tools should be considered as early as possible. It is advantageous when all firms cooperating within the same project use the same toolset. This leads to a common understanding of the applied methods and to exchangeability of software models. However, application of the same toolset does not guarantee that the various models of parts of the software can be simply integrated into one overall model. If this is needed or required, it should be clear from the beginning.
- When a source code management tool like CADESE is used, it is necessary that it is available during the whole project and on all computers where the software has to be compiled and/or modified. Only then it can be ensured that only authorised changes are made to the software.

Although some negative points have been mentioned in this paper, the final conclusion is that the application of CASE tools in this type of projects is an important contribution to the quality of the developed software.

10. REFERENCES

1. Maldari, P., et. al. ESA Support for the Italian SAX Astronomy Satellite Mission, ESA Bulletin, No 60, 1989, pp 53-58.
2. Dekker, G.J.; Hameetman, G.J., The development of flexible software for the Italian/Dutch satellite SAX, AIAA Computing in Aerospace 8, Oct. 21-24, 1991, Baltimore, Maryland. Also available as NLR publication NLR TP 91288 L.
3. Stuyvenberg, J.A. van; Voort Maarschalk, C.W.G. van der, Software flexibility ensured success of Dutch ANS and IRAS satellites, Proceedings of the ESA/ESTEC Seminar on Software Engineering, 11-14 Oct. 1983, ESA SP-199, pp 227-231.
4. Hameetman, G.J., Reliability and flexibility of the IRAS on-board software, Proceedings of the ESA/ESTEC Seminar on Software Engineering, 11-14 Oct. 1983, ESA SP-199, pp 215-220.

5. Baal, J.B.J. van, Hardware-software FMEA applied to airplane safety, NLR publication NLR MP 84073 U, 1984.
6. Kampen, S.; Kouwen, J., Design and development of the SAX-AOCS control software, in "1st ESA International Conference of Guidance, Navigation and Control Systems", Noordwijk, 4-7 June 1991, Proc ESA SP-323.
7. Nassi, I.; Shneiderman, B., "Flowchart techniques for structured programming", SIGPLAN Notices, aug. 1973, pp 12-26.

Discussion

Question D. NAIRN

Would your tools and techniques sclae up to a much larger project team?

Reply

The old manual methods allowed 2-3 persons to cooperate in software design. With automated design tools, this will scale up to 10-15 persons.

Note that to cooperate is meant to be : technical people working together on a design without imposing a management hierarchy on top to grant compatibility between different groups.

Experiences with the HOOD Design Method on Avionics Software Development

by
W. Mala and E. Grandi

ESG-Elektroniksystem- und Logistik- GmbH
P. O. Box 80 05 69
D 81605 München

SUMMARY

HOOD represents one of the most interesting approaches of the recent years to support an object oriented SW design for large embedded systems written in Ada.

This paper reports about experiences gained by the authors in the context of a current large european avionic developement project, where the Ada SW design has been performed using HOOD Version 3.0.

A simplified example describes the approach taken in the project for SW architectual design. A critical evaluation of the HOOD method follows, where advantages and disadvantages are discussed and some hints are given to overcome some identified weak areas.

The paper concludes with the recognition of HOOD as a promising approach and encourages further discussion to remove the weak areas.

1. INTRODUCTION

This report will give an overview on experiences gained with the HOOD SW-Design method during SW-Development for a large complex avionic-system.

Avionic-systems in general are complex, time-critical in their behaviour and have severe safety requirements, since malfunctions could affect human life and can cause lost of the aircraft. To fullfil the high quality, performance and safety requirements, a well defined, controlled development process must be established. This will include methods, tools and implementation languages.

For avionic software development Ada has been selected as implementation language and object-oriented methods are a most promising approach for the architectural design. In our project the HOOD method has been selected, which has been originally specifically developed for the 'architectural SW-design' of Ada SW-Systems.

We will describe the 'architectural design' approach, our experiences with HOOD and Ada and we will give some hints, which can, in our opinion, help to overcome some of the method's shortcomings.

The report will include the following areas:

- Project overview
- Introduction to the HOOD method
- the architectural design process
- Evaluation of the HOOD method
- Recommendations and conclusions.

Finally it should be emphasized that this paper reflects only the personal experiences and opinions of the authors.

2. THE PROJECT

The project is a large european avionic-system development, scheduled to be completed at the end of the nineties. The overall system has been subdivided into several subsystems, which are developed under responsibility of the participating nations.

The development process is based upon the DOD-STD-2167 life-cycle model, which has been tailored to the specific project requirements. The on-board target computers are multi-processors, based on the Motorola 68020 processor. The software has been developed in Ada with the DEC-Ada compiler on the host and the XD-Ada cross-compiler for the Motorola targets.

For the 'architectural SW-Design' the HOOD method version 3.0 (HOOD Reference Manual, Sept. 1989) has been used. The SW-Specifications have been produced

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

in a SADT-like form supported by a graphic tool and pseudocode for algorithms.

The HOOD experiences, referenced, are derived from development of a 50 K-SLOC SW-package.
The example used in the presentation, will show a simplified system-model, but nevertheless it is suitable for describing the HOOD experiences. The simplification will also effect the on-board target computer, were a single processor-system has been assumed, instead of the original multi-processor-system.

3. HOOD METHOD OVERVIEW

3.1 HISTORY

HOOD (Hierarchical Object Oriented Design) was first developed in 1987 as the result of a study contracted by ESA. Aim of the study was the definition of a method to support the architectural design for Ada in large embedded software development projects.

In 1989 the evaluation of the first user experiences led to a revision of the method as defined in the HOOD Reference Manual Issue 3.0. This version, almost fully tool supported, was used in our project. Further extensions to the HOOD definition are still under discussion.

3.2 DESIGN PROCESS AND SEMANTICS

Main features of the HOOD method are :
- top-down iterative process
- well formalised steps for design and documentation
- supporting techniques for object identification
- hierarchical software system model
- support for automatic code generation.

At the beginning the whole system is first considered as a root object and then decomposed into a certain number of so called child objects, which implement the parent's functionality. To implement this functionalities, the child object can call the operations of other child objects.

Therefore two kinds of relations can connect HOOD objects:
- 'include' relationship or 'parent/child' relationship
  A child object implements a part of the functionality of the parent and is graphically con tained in the body of the parent itself.
- 'use' relationship or 'senior/junior' relationship
  A senior object calls an operation of the junior object. An arrow connects graphically the senior and the junior object (fig. 3-1).

In the next design stage each child object becomes a parent and is decomposed into children of the following logical level. Each parent's operation is thereby connected through an 'Implemented_by' link with that operation of the child, which takes over the implementation of the operation functionality (dashed line in fig. 3-1).

The decomposition process is repeated for each identified child object, as far as their structure becomes so simple that they can be considered as terminal objects and directly implemented in Ada or pseudocode. The result is a hierarchical system model (HOOD Design Tree), which reproduces the include relationships between the objects (fig. 3-2).

A HOOD object has a name, an interface to the outer world and a body, which cannot be accessed from the outside.
There are several kinds of objects:
- *passive object*
  has no own control flow
- *active object*
  has an own control flow and reacts to external stimuli dependant upon its internal state through so called "constrained operations"
- *class object*
  an instantiatable template
- *environment object*
  an interface to another system environment
- *virtual object*
  can be associated with a physical processor

In order to decompose a parent into child objects, four activities are performed in each design stage, which represent the BDS (Basic Design Step):
- *Problem Definition*
  analysis of the context and SW requirements for the object
- *Development of the Solution Stategy*
  textual description of the functional implementation on a high abstraction level
- *Formalisation of the Strategy*
  identification of objects and operations and producing the HOOD Diagram
- *Formalisation of the solution*
  . formal definition of the object interfaces completing the Object Description Skeleton (ODS) for the parent

- filling in the interface fields in the ODS's of the children
- justification of the design decisions

Each step of the BDS is decomposed into more detailed subactivities.

In this way two relevant documents are produced for each object:
- a HOOD Diagram
- an ODS.

The ODS (fig. 3-3) consists of several fields, which contain a semiformal description of the object. The field OBCS (Object Control Structure) of terminal objects describes with Ada semantics the synchronisation of the operational flows and their interaction with the internal state of the object. The OPCS fields of terminal objects describe the sequential operation flows in Ada or pseudocode.

Additionally to the HOOD Diagram and the ODS, the HCS (HOOD Chapter Skeleleton) is automatically produced with the same structure as the BDS itself. The HCS represents the object documentation and can be used to produce the design documents.

The ODS contains all relevant information about the object and is source for the code generation process. The code generator converts
- objects in Ada packages
- OBCS's in tasks
- operations in procedures or functions.

Additional features of the HOOD semantics are:
- description of synchronisation properties for constrained operations (interrupts, timed_out, highly or loosely synchronised)
- description of main data flows
- description of exception flows.

4. ARCHITECTURAL DESIGN

4.1 PROBLEM DOMAIN

To simplify the presentation of the applied method, the problem domain has been reduced to a few essential functionalities.

The system is a combat aircraft, which searches for air targets, pursues, visually identifies, engages them and has a missile and a gun as weapons and a radar as sensor. A missile and a gun are the available weapons, the radar is the sensor. The pilot can perform a pure visual combat, but he can also require system support on the basis of navigation and radar information.

In our report, we will focus on the mission part of the system.

4.2 METHODICAL APPROACH

Structured development tools offer a practical way to convert conventional SADT-like SW specifications into an object oriented design.

In the Problem Definition Phase the SW requirements are restructured by applying the following methods and tools:
- DFD's (Data Flow Diagrams) for the description of data flows and transform processes
- STD's (State Transition Diagrams) for the description of the dynamic behaviour
- Data Dictionary for data description.

ERD's (Entity Relationship Diagrams) would be useful too for information modeling, but were not supported by our development tool. Basic strategies for object oriented design are applied:
- modeling real world objects at high logical system levels
- grouping DFD transform processes around the accessed data stores to support definition of abstract data types
- identifying object classes
  (in our case only parametric instantiation because of the target language ADA).

A context diagram is first drawn to analyse the environment beyond the system boundaries (fig. 4-1). It shows the real world as it appears to the system analyst and how the SW system is connected via a data bus interface with the subsystems:
- Cockpit
- Radar
- NAV
- Armament Control.

The DFD's and the STD's (fig. 4-2, 4-3, 4-4) complete the SW specification for the first abstraction level. The SW specification model focuses on the objects of the real world. The DFD (fig. 4-2) groups the main processes and data flows around the connected physical subsystems or around cockpit displayed items, such as
- list of the designated targets
- steering cue.

The STD (fig. 4-3) shows the macro states of the system, is disjunct to the DFD and contributes directly to identify design objects and operations. The STD (fig. 4-4),instead,is the control process of the DFD in fig. 4-2 and is to be balanced with it.

The following considerations prove the balance:

- *Radar Monitor* and *Radar Correlation* are continuos processes and do not appear in the STD.
- *Radar Control*, *System TL Update* and *Pilot's TL Update* are continuos processes too, but they control the start/stop of *Steering Calculations*, *Missile Envelope*, *Missile Control* and *Gun Calculations*.
  Therefore they are expected to appear in the STD for generating the correspondent events.
- The events *Weapon Selected* and *Weapon Deselected* are generated by the terminator *Armament Control*.
- STD actions correspond to DFD processes or parts of them.

Object candidates and operations are identified from the the specification model. Crucial items for object identification are terminators and data stores in the DFD's and states in the STD's.

Fig. 4-5 highlights the objects candidates RADAR, TARGETS, STEER_CUE and WEAPONS by enclosing the corresponding DFD areas into a dashed frame. The data stores, the data dictionary and the data flows provide information about the object state and the contents of the exchanged messages. Since we already developed our essential strategies for an object oriented design during the Problem Definition Phase by restructuring the SW specification, in the next design stage, Development of the Solution Strategy, we simply produce a text to explain the models.

The result from the Formalisation of the Strategy is shown in the HOOD diagram in fig. 4-6. The objects RADAR, WEAPONS, TARGETS and STEER_CUE represent entities of the real world and correspond to object candidates of the solution strategy. The INIT_STATE object represents a system macro state we identified with the STD. The BUS object is a model for the bus interface from the context diagram.

Constrained operations are marked by a trigger arrow on the HOOD diagram. They were identified by analysing system synchronisation issues. The identified constrained operations determine the active status of the objects and implicitely the processes of the system.

The operations INIT_STATE.power_on, INIT_STATE.power_off and BUS.get_bus_data are designated as ASER_by_IT constrained operations, because they represent system interrupts activated by the external events power on, power off and new bus data available.

Operations which have to preserve the consistency of accessed data structures are to be designated as constrained. This is the case, for instance, for TARGETS.designate_trgt and TARGETS.update_list, which access the list of the designated targets following the pilot's demands or target information passed by the radar respectively.

Other operations are identified to be constrained, because of their concurrent nature with other system operations. This applies to the constrained operations of STEER_CUE and WEAPONS, which implement concurrent algorithms.

Operations, which run inside the control flow of other operations, are designated as passive operations. Therefore all the init operations (except BUS.init) are passive, because they run sequentially in the control flow of INIT_STATE.power_on.

The operation BUS.det_output_data converts the system data into the bus format and is called whenever new system data have been calculated. The BUS object contains a cyclic background process, which transmits the output data to the bus and raises a bus error exception, if no bus data arrive within a predefined time interval. This background process can be considered as the cyclic portion of the operation BUS.init. Consequently the bus error exception is propagated outside the system environment via INIT_STATE.power_on.

The following subfunctions in the BUS object:

- receiving raw bus data
- cyclic bus data output
- raising bus error exception

can be reused in other avionic subsystems. Therefore it is sensible to define a BUS_CLASS object, which can be instantiated by passing constant parameters for the bus protocol (minor cycles, message sizes, memory addresses, etc.). In the level 2 design stage the BUS object is expected to include an instance object of BUS_CLASS.

Finally with the Formalisation of the Solution Strategy the ODS of the identified objects are filled in. The OBCS of the BUS object could look like as follows:

```
accept INIT do
--    call internal operation
      loop select accept GET_BUS_DATA
      (BUS_DATA:BUS_DATA_TYPE) do
--    call GET_BUS_DATA OPCS
      end GET_BUS_DATA;
      or
      accept DET_OUTPUT_DATA(
      OUTPUT_DATA: OUTPUT_DATA_TYPE) do
--    call DET_OUTPUT_DATA OPCS
      end DET_OUTPUT_DATA;
      or
      delay 0.02;
--    if the stop flag was set exit the loop;
--    if a bus error was detected raise exception
--    call internal bus output trigger operation
         end select;
         end loop;
         end INIT;
```

The justification of the design decisions concludes the first Basic Design Step. For each identified child object a further BDS is run.

The refinement process is carried on till the internal object structure needs no further decomposition and the identified operations consist of simple sequential flows, which can be easily implemented in ADA or pseudocode.

After the design has been converted into Ada code by the codegenerator, the programmer completes the source code by adding for instance representation clauses, task priorities and by substituting pseudocode fragments with Ada code.

5. EXPERIENCES

5.1 HOOD ADVANTAGES

The advantages of HOOD, in accordance with the intention of its developers, are certainly to be found in the support of the design of large Ada systems, especially when development teams are located at different sites and main system components have to be subcontracted.

The well defined process steps and the contemporary production of a design document, which is structured as the development cycle itself, enhance standardisation, support information exchange and insure a transparent design process. The automatic conversion of the HOOD design into an executable Ada program alleviates the developer's work and reduces the skill discrepancy between experienced and less experienced personnel, which has often a negative impact on team synergy and product's consistency in large development efforts.

The use of HOOD requires that the programmer only needs to program in the small; module and process structures are derived directly from the HOOD design. The strict top down process of HOOD is a powerful feature for gradually reducing system complexity by decomposition and partitioning into separate work packages.

On each system abstraction level the decomposition process leads to a refined object structure, however this is achieved through a kind of functional mechanism such as the 'Implemented-by link'. This can help to overcome the gap between SW specification and object oriented design, if a complete functional specification is available as input for the design phase.

HOOD enables the easy construction of an executable system model on each abstraction level and supports herewith early prototyping.
The hierarchical system model supports two complementary test strategies during SW Integration:

- Top down test using SW stubs
- bottom up test starting from tested terminal objects.

5.2 PROBLEM AREAS

5.2.1 SW REQUIREMENTS / OOD TRANSITION

Converting a pure functional SW specification into an object oriented design is a fundamental problem. If the SW specification was produced following a conventional function and data model, some additional problems will arise:

- the flat structure of the specification model is to be converted into a hierarchical one
- data and functions have drifted apart
- system's states and dynamic behaviour are not easy to identify in the SW specification model.

We feel that a restructuring of the SW requirements by means of hierarchical DFD's and STD's has provided a suitable basis for the HOOD design. The DFD's were able to achieve the necessary abstractions, to bring data and functions together and could be easily derived from the specification model, because of similar graphics and semantics.

Deriving STD's was more difficult, because although the specification model could express process sequencing, it highlighted relevant moding information insufficiently. Even though the original functional specification could be successfully converted into a well structured object oriented design, some disadvantages for requirements traceability had to be accepted.

After the design was completed, about 20% of the SW requirements could not be clearly related to a design object; they were marked as "partly fullfilled" in several affected ODS's.

5.2.2 REAL-TIME ASPECTS

If we consider that HOOD was intended to support ADA and that ADA addresses embedded systems, the insufficient real-time semantic must be regarded as a major problem with HOOD.

We believe that HOOD diagrams and ODS's do not express sufficient system behaviour and miss particularly a representation of operation call chains together with their activating events. Structured-charts-like diagrams could be a suitable tool for representing operation call chains and convey additional control flow information for storage in the tool database.

HOOD's support for identification of processes in the system is, in our opinion, also insufficient, because clear guidelines for the usage of the concepts
- functional constrained operation
- constrained operation by type of execution request
- active and passive objects

are missing in the Reference Manual.
Little support was found for dead-lock analysis and estimation of response time.

5.2.3 CODE PRODUCTION

Mapping the design into Ada code is in our opinion another problem area of HOOD. The code generation does not produce any additional execution overhead, but destroys the hierarchical design structure.

The terminal and non-terminal design objects appear in the code as a flat landscape of withed Ada packages, which cannot be clearly mapped back into the original hierarchy. The position of the 'with-clause' before the specification or the body of the Ada package was, in the HOOD version 3.0, no longer a clear hint for distinguishing a 'use' and an 'include' relationship. The gap between design and code structure obstructs reverse engineering processes.

In addition, the HOOD semantic does not exploit some useful features of the Ada language, for instance task types, entry families, nested constructs and access types (even though the latter are no desirable features for safe critical systems).

The structural gap between design and code and manual extensions of the generated code produce negative impacts on SW test and maintenance. Copying manual code extensions back into the ODS for consistency was felt to be a duplicated action. Maintaining consistency among several versions of code files and ODS's was sometimes a tedious and error prone process.

6. RECOMMENDATIONS

Recommendations will be addressed in two directions, to the
- HOOD Users
- HOOD Technical Group, which is responsible for the method definition and enhancements

HOOD Users:
Recommendations addressed to the HOOD user are:
- take real world objects as a basis for SW specification, avoid separation of functions and data. Objects must encapsulate functions and data, if both are separated in the SW specification the identification of the objects will be very difficult.
- Proceed iterative for SW requirements analysis and architectural design,
- make use of prototyping in early phases to investigate dynamic system behaviour.

In our days most of the Project models used for SW-Development are based on the 'waterfall model', where the SW-design can only be initiated after the SW-specification has been completed and accepted. There are many reasons for this approach, because system and software knowlegde for such complex system may reside in different specialists teams located on different sites. Nevertheless we believe that a top down iterative development process (spiral model) for SW requirements analysis and SW design would improve the development process.

The use of (rapid) prototyping in early phases of the project as supported by HOOD will ensure, that the dynamic behaviour of the system will be in accordance with the requirements.
- fill in Object Definition Skeletons (ODS) with Ada code as far as possible.

This will ensure that the SW design and the Ada code will be more consistent and maintainable and will support requirements traceability.

HOOD Technical Group:
Based on our experiences the following HOOD enhancements would be desirable:

- In order to enhance the real-time capabilities, we would suggest extending the HOOD semantic with a graphical representation for operation call chains. The control information should be stored in the database and could also be used to support the identification of circular dead-locks.
- A processing time analysis mechanism for object operation could be utilised for supporting an early analysis of the system reactions, particularly in conjunction with prototyping.
- A mapping of the include relationship into nested Ada packages could be, in our opinion, an interesting approach to overcome the structural gap between SW design and Ada code for shallow system architectures.

These real-time enhancements would also contribute to the architectural design, for instance, by supporting the identification of additional required synchronisation objects and the active/passive properties of the objects.

7. CONCLUSIONS

The HOOD Version 3.0 has some shortcomings and weak areas, however we recommend HOOD as a very promising method for 'SW-Design' in large Ada projects.

HOOD's well defined formalised steps, the automatic conversion of the design into Ada code and the automatic production of design documentation improves the productivity during development, enhances information exchange and re-useability significantly.

References

Ada Language Reference Manual, ANSI/MIL-STD 1815A
HOOD Working Group, HOOD Reference Manual Issue 3.0 (European Space Agency, 1989)
Robinson, Peter, Hierarchical Object-Oriented Design (Prentice Hall International, 1992)
Booch, Grady, Software Engineering with Ada, Second Edition, (Benjamin/Cummings, 1986)
Coad and Yourdon, Object-Oriented Analysis (Yourdon Press, 1991)

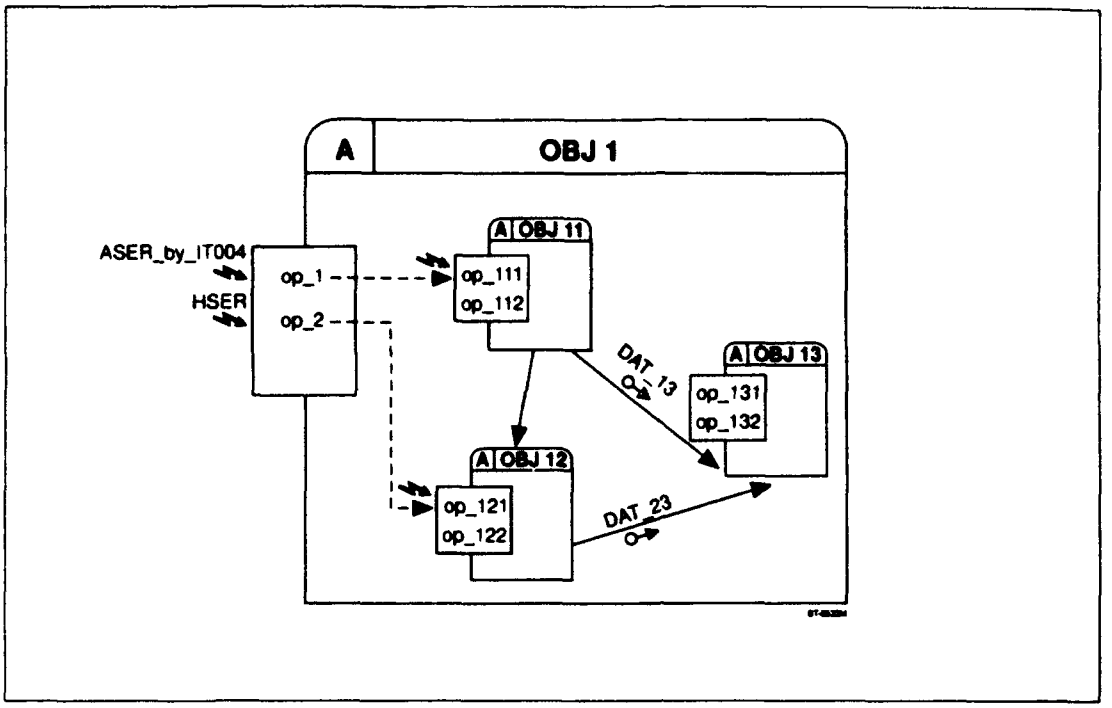


Fig. 3-1 HOOD Diagram

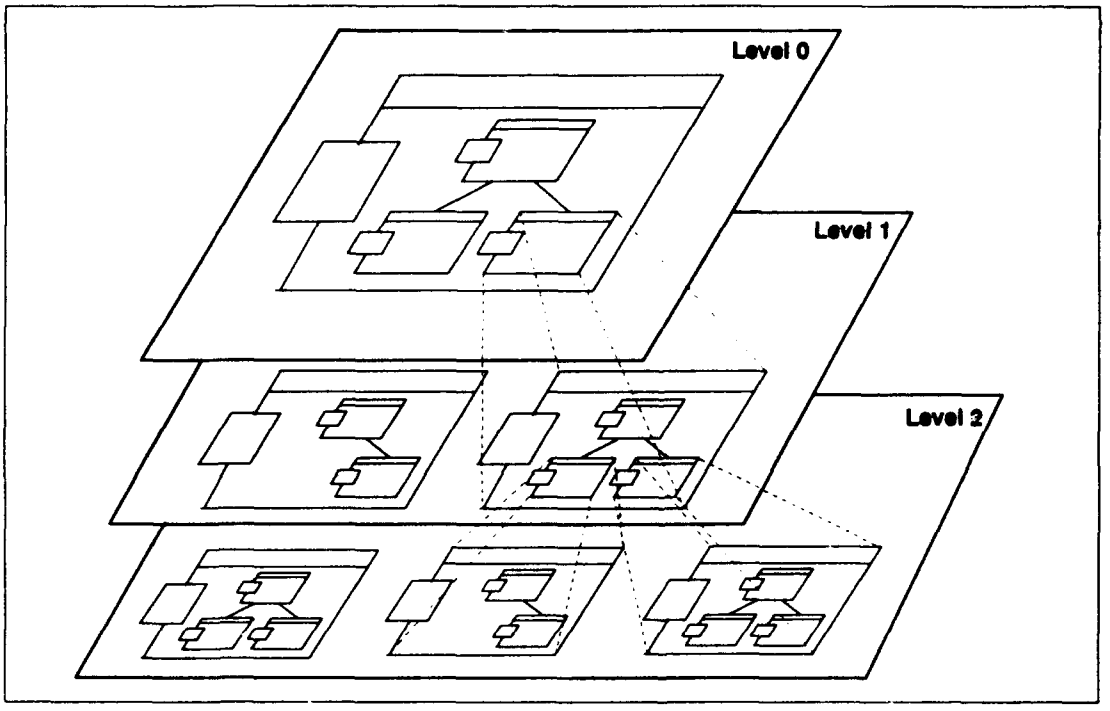


Fig. 3-2 HOOD Object Hierarchy

OBJECT Object Name IS [CLASS] Object Type

DESCRIPTION

IMPLEMENTATION OR SYNCHRONISATION CONSTRAINTS

PROVIDED INTERFACE
(Types, Constants, Operations, Exceptions,...)

REQUIRED INTERFACE
(Objects, Types, Operations, Exceptions,...)

DATAFLOWS

OBJECT CONTROL STRUCTURE
(Description, Constrained Operations, Code or Implemented-by)

INTERNALS
(Objects, Declarations, Operations)

OPERATION CONTROL STRUCTURE
(Name, Used Operations, Exceptions, Code, Exception Handlers or Implemented-by)

END-OBJECT Object Name

Fig. 3-3 Object Description Skeleton

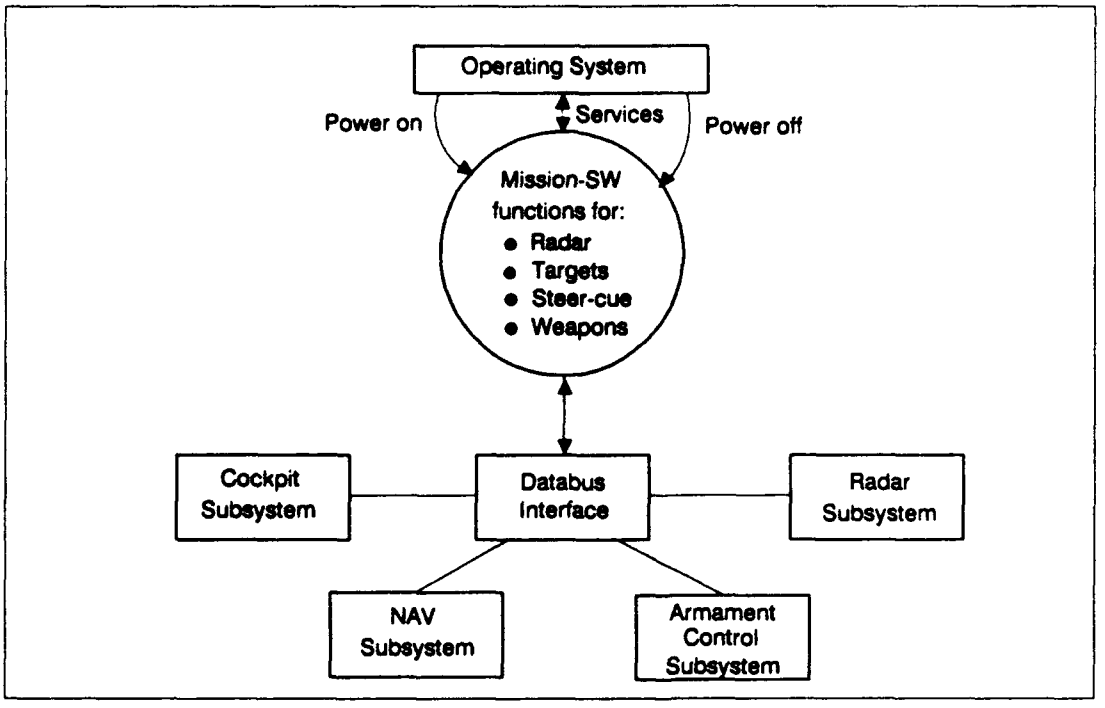


Fig. 4-1 Context Diagram

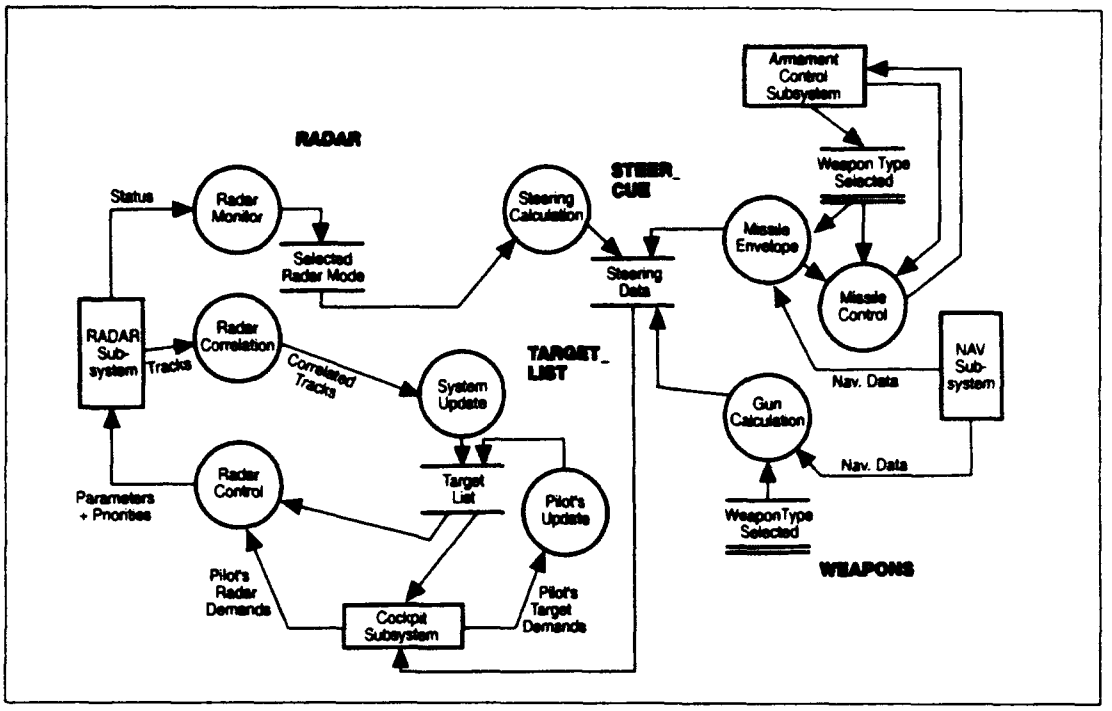


Fig. 4-2 Data Flow Diagram level 1

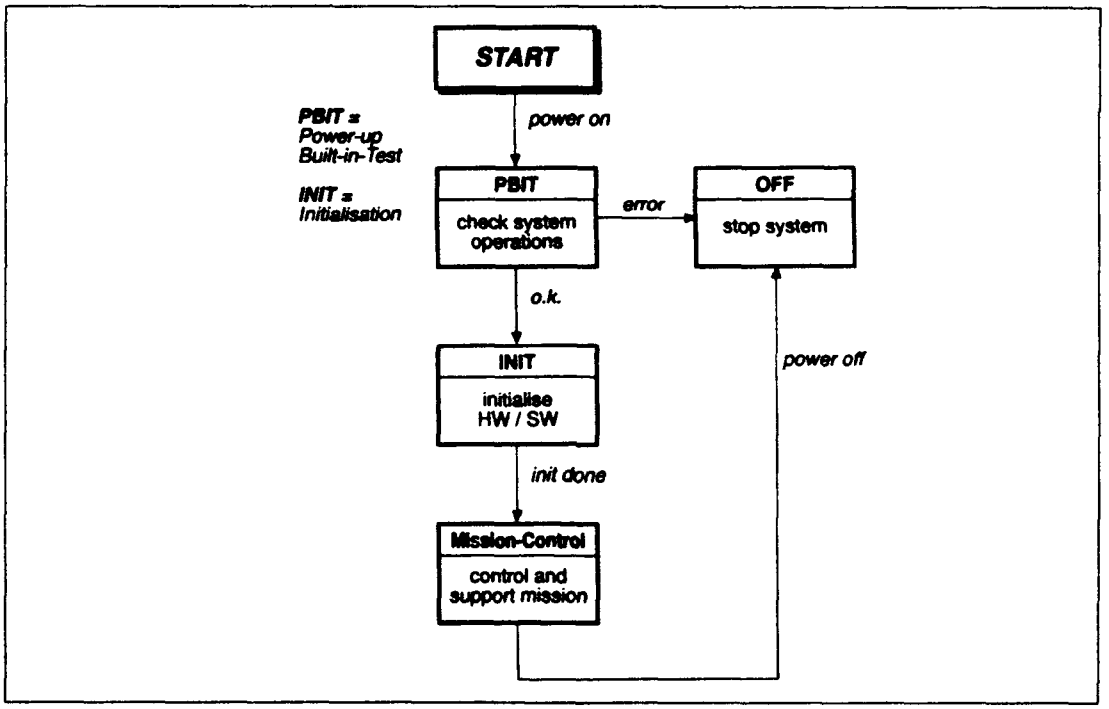


Fig. 4-3 State Transition Diagram level 0

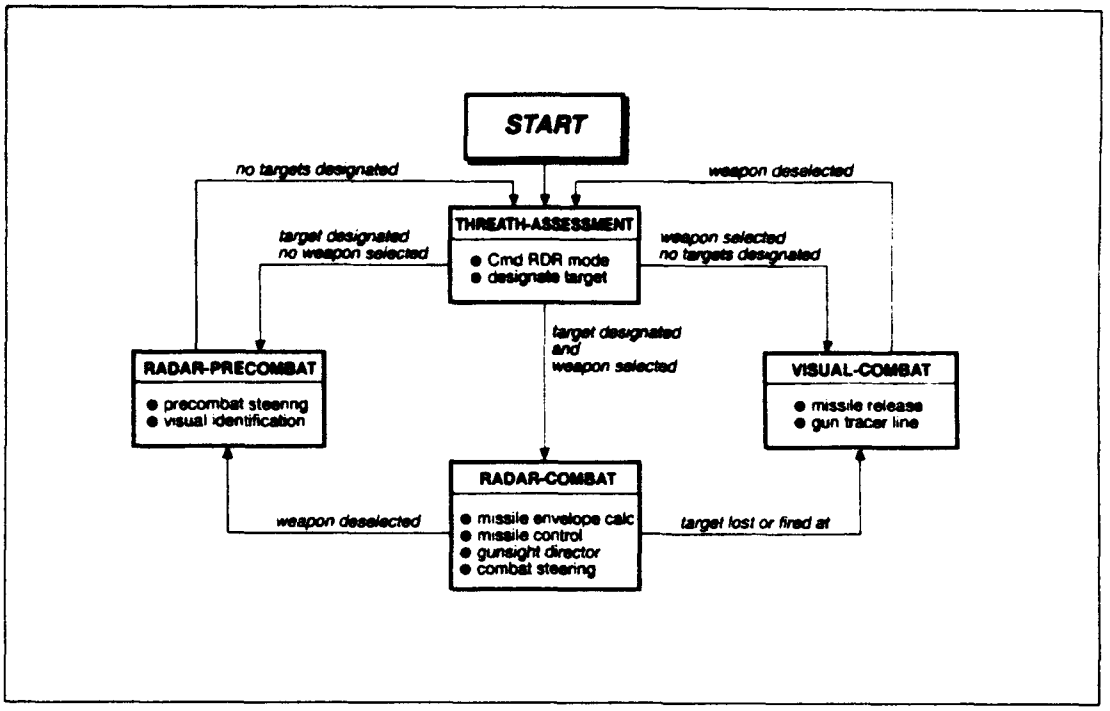


Fig. 4-4 State Transition Diagram level 1

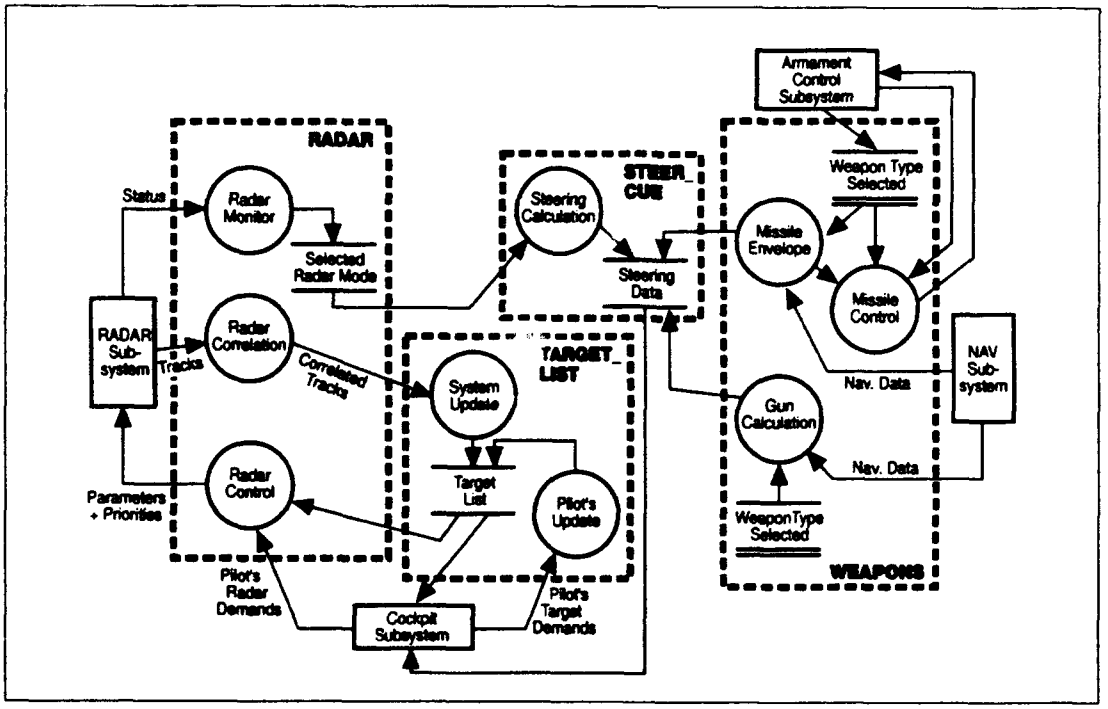


Fig. 4-5 Object Candidates

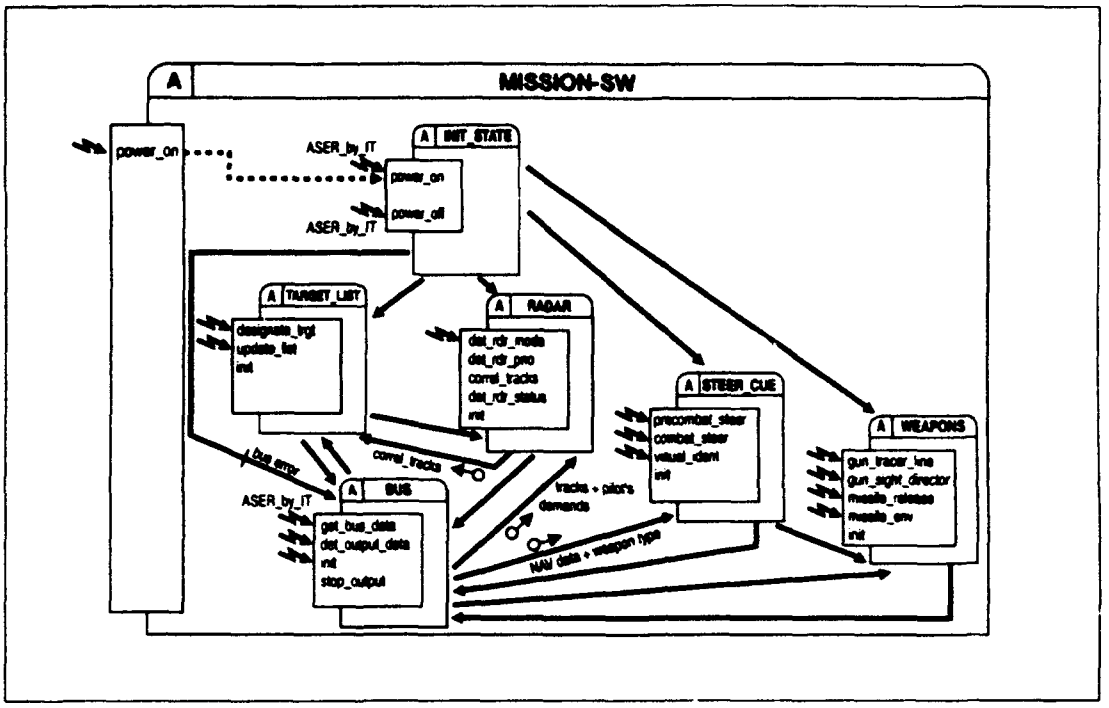


Fig. 4-6 HOOD Diagram for Mission SW

Discussion

Question H. LE DOEUFF

What was the real time structure used? Was the correspondence of one active object = one Ada task respected? What were the real time performances?

Reply

The real time structure has used several Ada tasks, defined as "active objects". There were no problems with the recognized "real time" performance.

SOFTWARE ENGINEERING METHODS IN THE HERMES ON-BOARD SOFTWARE

Philippe Lacan
AEROSPATIALE - Espace & Défense
66, route de Verneuil - BP 2
78133 Les Mureaux Cedex
France

Paolo Colangeli
DATAMAT -Ingegneria dei sistemi SPA
Via Simone Martini, 126
Roma (EUR) 00142
Italy

ABSTRACT

The HERMES On-board Software is executed in a complex multi-processor environment composed of two segregated computers pools. It provides all services to control the trajectory and the attitude of the spaceplane and its configuration. It also provides aids to the On-board Crew and the Ground Facilities in the Mission and Space Vehicle management.
Being highly "Safety Critical", the attitude control functions are foreseen to be supported by a quad-computers pool in hot redundancy, running in parallel and tightly synchronized.
To support "Mission Critical" functions, as mission and vehicle management, a dual-computers pool in cold redundancy is the baseline.
This paper describes how the use of HOOD methodology has been experienced in the HERMES On-board Software Mock-Up framework, from now onwards designated for the sake of brevity as MU.
In this technical survey of MU, a number of figures characterizing application size, specification and design complexity of the developed software along with a technical balance of the experienced methods are given.

Keywords: Software, Guidance-Navigation and Control, Design Methods, HOOD, SADT, Metrics

1. INTRODUCTION

1.1 HERMES Spaceplane In-Flight Mission

The HERMES Spaceplane is part of the European program to support "man in space" activities. It aims to be a transfer vehicle for servicing space stations (e.g. COLUMBUS) with human operators.
According to this general objective, the "in-flight" mission of the spaceplane may be defined with three main phases:

- the "launch" phase, where HERMES acts as the "equipment bay" of the ARIANE 5 launcher in order to control the launch trajectory;
- the "orbital" phase, after ARIANE 5 jettisoning, where HERMES is like a "satellite" and has to perform attitude and orbital control as well as rendez-vous operations;
- the "re-entry" phase, after desorbitation orbital arc, where HERMES becomes and hypersonic glider.

1.2 Data Processing Organization

In order to support the above described functions, the Spaceplane is designed to around a centralized Data Processing system. Data Processing system is organized in two segregated computers pools as shown in Figure 1:

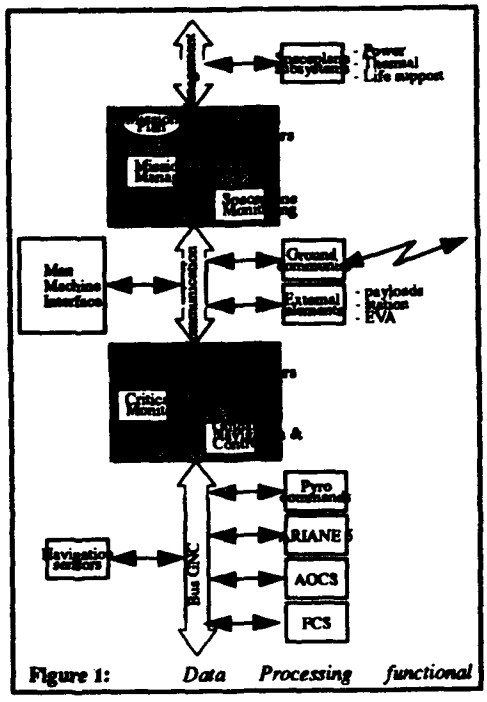


Figure 1: *Data Processing functional*

- the GNC pool supports all "safety critical" functions of the spaceplane mainly related to flight control;

- the MMC pool executes "mission critical" functions of the spaceplane as: mission plan execution and monitoring of the vehicle consuma resources, but is never involved in safety operations.

The Mission pool is composed of two computers, which can manage the spaceplane subsystems (Power supply and distribution, Thermal control, Life support) through two cross-strapped numeric buses using standard interface boxes, as shown in figure 2. One computer is active and controls the subsystems, the other one plays a monitoring role during the critical phases of the flight (launch and re-entry) and may supersede the first one in case of failure. All accesses to physical items inside the subsystems are redundant as well as numeric buses control.

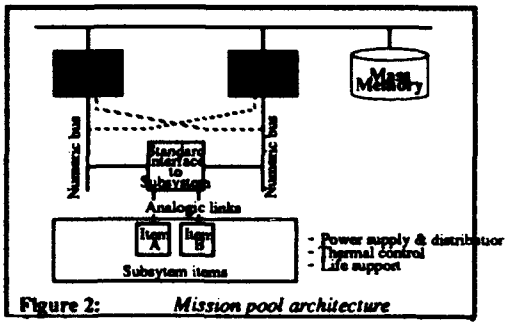


Figure 2: *Mission pool architecture*

As depicted in figure 3, the GNC pool is composed of four computers interconnected via a specific Inter Processor Network (IPN). Each computer controls its own numeric bus to access navigation sensors and flight control actuators: ARIANE 5 equipment for launch, AOC System on orbit and Flight Control System for atmospheric re-entry. During the critical phases of the flight, the four computers are running in parallel the same software, they exchange the inputs data and commands via the IPN to mutually monitor the health of each other. The GNCs computers are synchronized thanks to a specific protocol on IPN.

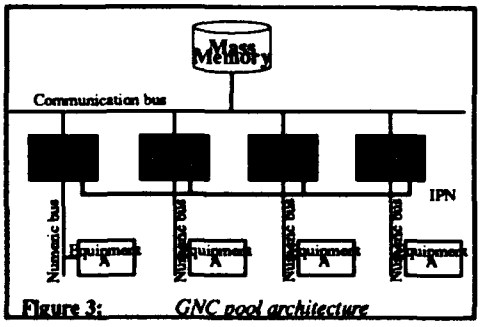


Figure 3: *GNC pool architecture*

1.3 HERMES On-Board Software Functions

The functions supported by the On Board Software are the following:

- to acquire and execute the mission plan in close cooperation with crew and ground. The mission plan is organized in "sequences of operations" to perform with regard to predefined events;
- to control and monitor the execution of "sequences of operations" defined in the mission plan, while insuring a safe status of the spaceplane and monitoring the levels of consumable.
  The "operations" generally correspond to set the subsystems in a given configuration, then to perform mission-dependent operations (Pay-loads, Extra Vehicle operations) with associated flight control objectives;
- to perform the flight control operations according to the different flight phases and associated control means:
  - -- ARIANE 5 actuators for the launch phase;
  - -- Attitude and Orbit control thrusters for the orbital phase, rendez-vous operations and desorbitation arc;
  - -- cold gas thrusters for the ending phase of rendez-vous (in proximity of the target);
  - -- flight control aero-surfaces for the atmospheric re-entry;
  - -- breaks and nozzle wheel during the landing roll.

The mission pool is in charge of all "mission dependent" operations which are not safety critical. The dual redundancy allows to cover one failure, assuming a failure detection and recovery by the crew (Fail Operational).

The safety critical functions of flight control and associated equipment management are performed by the GNC pool, which is designed to be failure tolerant by implementation of FDIR mechanisms (Failure Detection Isolation and Recovery). Thanks to its quad redundancy and a majority vote mechanism, the GNC pool is able to tolerate two failures (Fail Safe/Fail Operation).

In addition, the GNC pool is able to initiate and perform autonomously the re-entry operation in case of crew incapacity, lost of ground communications or lost of the mission pool.

For the GNC software, the technical challenge is threefold:

- firstly, to meet the safety requirements;
- secondly, to satisfy a number of very tightly real time constraints (deadlines) in such a complex (quad, synchronized) mode;
- lastly, to comply with processing power and memory size limitations due to space environment (the software is to be executed by a SPARC monoprocessor board in each computer).

1.4 HERMES On-Board Software Development Industrial Organization

In addition to the above mentioned technical constraints, the HERMES On-Board Software development has to face with a complex industrial

organization involving several companies specialized in the space software field:
- AEROSPATIALE as Prime contractor
- DASSAULT AVIATION for Atmospheric flight control software
- MBB for Orbital flight control software
- MATRA for System software
- DATAMAT for Mission management software
- TRASYS for Launch flight control software.

In order to assess the feasibility of the HERMES On-Board Software development and to experience the use of software design techniques, we start, three years ago, a "Mock-Up" process involving the main participating companies.
The present paper gives a synthesis of the results of this process especially the experience of using software engineering methods and tools: SADT, HOOD and the Ada language.

2. MOCK-UP PROJECT BOUNDS

2.1 Purpose of the Mock-Up

At the beginning of C1 phase, in 1988, the development and exploitation of the MU was intended to provide aids in the verification of the HERMES On-board Software (HOSW) specification.
The MU was mainly aimed, on one side, at providing support for the evaluation of technical choices in terms of: software architecture; trade-off between performances and functional requirements; ADA Run-Time needed features, and, on the other side, at exercising the software development methods and ADA language to be adopted in the HOSW life cycle.

2.2 Organization of the Mock-Up

The MU is split in "Application SW" and "Execution Environment".
The "Application SW" MU consists in a number of components modelling five major functions of HOSW, namely: Localisation, Mission Management and GNC for Launch, Orbital and Re-entry phases.
The MU "Execution Environment" includes, as software components, the Support System Software and the System Software (2nd Level), hosted by the corresponding MU "Execution Environment" hardware components, that is, the Support System and the Core System.
In the following, a brief overview of each MU Application SW component is given in order to better bound the project context.

1. LOCALISATION application software (LOC)
It simulates the localization function during the orbital phase of the Hermes flight, in nominal equipment mode. It includes a system management function of the GNC.
The functions performed by the localisation software are:
- calculation: ensures the simulation of localisation calculations to compute the absolute attitude and the absolute position;
- guidance;
- acquisition: ensures the acquisition of the cyclic data generated by the simulated localisation equipments (IMU, SST, GPS) and by the other simulated GNCs (for the voted attitude). It enables data monitoring too;
- sending: controls the transmission of the generated messages to the Support System simulating the external environment;
- management: interprets and executes the commands and tele-commands provided by the external environment. It sends a cyclic global report.

2. MISSION application software (MMU)
It implements the following four major functions:

- Operations Plan Execution
- Vehicle Subsystems Management
- System Configuration Database Management
- Anomalous conditions handling

The Operations Plan contains a description of the activities to be performed at run-time: it describes each operation in terms of elementary actions and scheduling time.
A generic elementary action is usually accomplished by interacting with the components external to the MMU through the simulated 1553 Bus, according to specific communication protocols.
The external components interfaced with the 1553 Bus are:
- Generic Vehicle Subsystem (#1 and #2);
- Operator Interface (PPT);
- GNCL Mock-Up

The transactions on the 1553 Bus are organized in strictly deterministic policy: the whole bus activity is mastered by one Bus Controller (BC, the Mission MU) and all other connected devices play the role of Remote Terminals (RT) being polled by the BC. The software implementation of MMU guarantees the correct processing of all I/O transactions on the Bus.

3. GNC-Launch Phase application software (GNCL)
It reproduces the functional characteristics of the Ariane 4 Guidance, Navigation and Piloting functions during launch phase:
- the acquisition of data from BGY and IMU;
- the navigation, guidance and piloting;
- the launch flight sequence management;
- the telemetry stream generation.

The execution of these application components is scheduled according to the timeband of the component: the timeband defines points regularly spaced in time at which the component must be triggered (triggering points).

4. GNC-Orbital Phase application software (GNCO)
The functional capabilities of the MU are given in the following:

- guidance: the actual and the desired orbit are compared. After checking if both orbits intersect and if both orbits lie in the same orbital plane, different strategies can be envisaged.
- navigation: relative navigation is implemented. The output of a simulated inertial measurement unit is evaluated. The gyro measurements (strapdown system) are integrated. Quaternions are used. Accelerometer data are converted into velocities and position of the spaceplane.
- Orbital Flight Control: actual and desired attitude are compared, the errors are calculated and the desired action is determined by a phase plane controller system.

5. GNC-Re-entry Phase application software (GNCR)
It implements the following functions:

- acquisition and processing of data from GPA sensors;
- guidance during hypersonic and terminal flight;
- piloting during atmospheric re-entry phase;
- spaceplane stabilisation during atmospheric re-entry phase.

As far as the MU "Execution Environment" is concerned the software components are:

1. System Software (2nd level)
It provides the application software MUs with the services they need to properly run.
The offered services, from the functional point of view, are:
- scheduling services
- communication services
- time management services
- monitoring services
- interrupt handling services

2. Support System Software
It provides software tools allowing the application software MU to be experimented, that is, the application software MU is made free of consuming an input dataflow, previously defined in a scenario, and producing an output dataflow, properly logged to allow its following analysis.

As far as the MU "Execution Environment" is concerned the hardware components are:

1. Core System
It represents the V3 On-board Data Processing architecture, that is the target machine, and mainly consists of two Application Processors and a standard VME bus, which links them to each other and to the Support System too.

2. Support System
It consists of one processor performing the on-line functions in terms of simulated input load to the APs and, the other, the on-line processing of output data generated by the APs. Both the processors are connected to the VME bus of Core System.

3. MOCK-UP PROJECT FIGURES

This chapter is aimed at giving measurements of size and complexity of the whole application software mock-up, that is, LOC, MMC, GNCL, GNCO and GNCR.
A more user friendly mechanism of associated icons is adopted to increase paper readability and understandability.

3.1 Size of the Application

Figure 4 gives, in a synoptic way, some figures about the size of ADA programs, constituting the LOC, MMC GNCL, GNCO and GNCR components of the MU, and related project documentation.
The statement delimiter (i.e. ";") were taken into account in computing the Ada statements total number

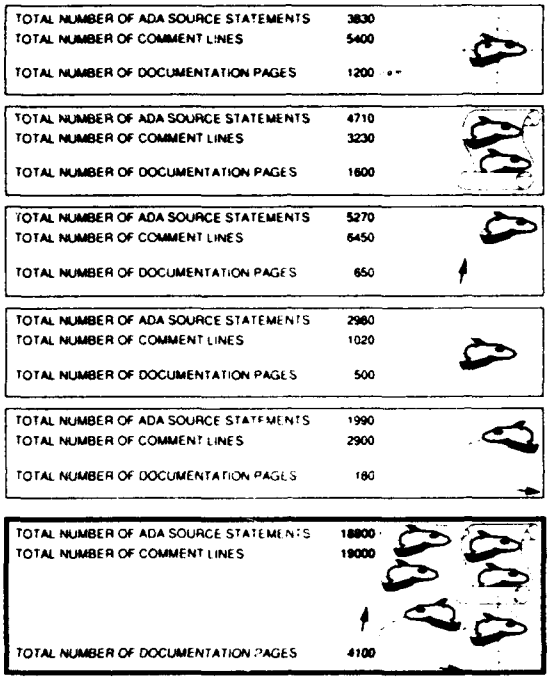


Figure 4: *Size of respectively LOC, MMU, GNCL, GNCO and GNCR components of application Mock-Up*

3.2 Metrics of Design Phase Complexity

Complexity does not inevitably mean lack of quality, nevertheless a complex design demands much more time for verification and testing. Generally speaking, the more complex software design is, the more exacting validating and coding activities are. In the following, every element, which can be considered as

representative of design phase complexity, is presented.

3.2.1 Design Tree Measurements

Figure 5 starts the navigation through the metrics of design complexity by providing the total Number of Objects and the Maximum Depth values.

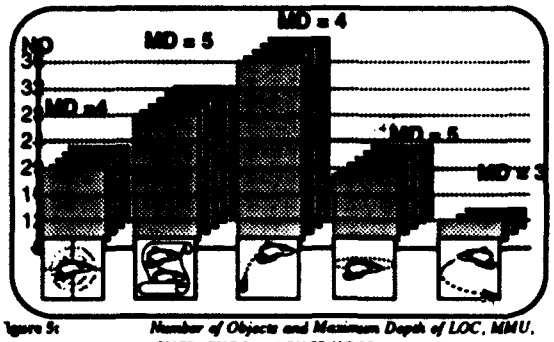


Figure 5: *Number of Objects and Maximum Depth of LOC, MMU, GNCL, GNCO and GNCR HOOD model*

NO: Number of Objects
It should be considered only a dimensioning parameter, without any other direct complexity implication, since both cases are possible: a design with a lot of very trivial objects and, on the other side, a design with very few objects, but extremely complex.

MD: Maximum Depth
A decomposition level is added every time that an object, estimated not enough detailed, needs to be further split into child objects.
The design complexity depends on that value, which only gives information about the deepest branch of the tree, without any other indication about the complete arborescence, being the depth of other branches unequal.
Generally, the higher Maximum Depth value is, the higher the number of *implemented_by operations* is.
The *implemented_by* relations, increasing the number of links and information conveying, make the design more and more unreadable and understandable.
It is worth pointing out that, contrary to one could think and expect, the three GNC mock-ups design trees sensibly differ.

3.2.2 HOOD Objects Metrics

In this section definition and meaning of metrics, which were calculated for each HOOD object componing the architectures, are given.

DF: Dependency Factor
The complexity of a design increases depending on the links generated by the USE relation.
The Dependency Factor takes into account the fact that a modification of an used object can consequently involve other modifications in the using objects.
The higher the DF value is, the harder the management of propagation of involved modifications is (waterfall modifications).
The Dependency Factor of the O_i object is calculated by the summation of the DFs of all the objects used by O_i with the convention of arbitrarily setting to 1 the DF of objects which do not use any other object.

NP: Number of functions or Procedures PROVIDED by an object
The understanding of the interface offered by an object is mainly due to the operations declared as PROVIDED.
The intrinsic complexity of an object depends on the number of operations provided by both its terminal and not terminal objects.
Infact an operation declared at several levels as IMPLEMENTED BY increases the design complexity, even if there will be a unique procedure body, at terminal object level, implementing that PROVIDED operation.
The NP of the O_i object is calculated by the summation of the NPs of its child objects and where NP is equal to the number of PROVIDED operations for terminal objects.

COU: COUpling factor
HOOD methodology recommends for designing an object with a number of objects using it higher than that of used objects.
The Coupling factor of O_i rightly expresses the ratio between the number of objects using O_i and that of objects used by O_i.
The numerator and denominator of the ratio are conventionally set to 1 in case of respectively no object using O_i and no object used by O_i .
The Coupling factor normally could be significantly higher than 1.

NT: Number of Terminal objects
Finally, the Number of Terminal objects were measured. If the object O_i is a terminal one, then NT is conventionally set to 1, otherwise NT is calculated as the summation of the NTs of the child objects of O_i .

Tables 1, 2 and 3 summarize the HOOD objects metrics for each application software mock-up.
It could be of some interest verifying that while the HOOD theory, recommending for designing an object with a number of objects using it higher than that of used objects, should lead to values of coupling factors significantly higher than 1, on the contrary, the applications show weak coupling factors values, ranging from 0.12 to, exceptionally, 10.

MMU Object	DF	NP	COU	NT
MMU	1	46	1.00	18
PPT	1	2	3.00	1
GNC	1	2	2.00	1
TELEMETRY	4	2	1.00	1
SUPERVISOR	1	3	3.50	1
BUS_STATUS	1	3	10.00	1
OPERATIONS_PLAN	10	9	0.40	3
CTRL_OPERATIONS_PLAN	10	4	0.33	1
OP_SCHED_LIST	1	2	1.00	1
OP_SEQUENCER	8	3	0.25	1
SS_1	3	25	1.00	10
PRESET_PROTOCOL	3	13	1.00	5
PRESET_REQUEST	2	2	0.50	1
STATUS_MESSAGE	9	2	0.14	1
CONTROL_MESSAGE	2	3	0.50	1
PRESET_PROTOCOL_STATUS	1	4	3.00	1
SS_STATUS	1	2	2.00	1
COMMAND_PROTOCOL	1	5	1.00	1
SYSDATA_PROTOCOL	2	1	0.50	1
SENSOR_PROTOCOL	5	6	0.33	3
MONITORED_PARAMETERS	1	3	1.00	1
SENSOR_DATA_MESSAGE	7	1	0.20	1
CURR_PARA_VALUES	1	2	1.00	1
E_SYSTEM_SOFTWARE	1	6	4.00	3
E_BUS_1553	1	4	1.00	1
E_TIME	1	1	1.00	1
E_TELEMETRY	1	1	1.00	1
E_SCHEDULER	1	1	1.00	1

Table 1: *LOC and MMU HOOD objects metrics*

LOC Object	DF	NP	COU	NT
P_LOC_MOCK_UP_MGR	4	10	0.33	1
P_LOCALISATION	1	36	2.00	7
P_LOC_MANAGER	3	6	0.50	1
P_LOC_EQUIPMENTS	1	20	4.00	4
P_EQUIP_CTRL	3	5	0.33	1
P_IMU	1	5	1.00	1
P_GPS	1	5	1.00	1
P_SST	1	5	1.00	1
P_ORBIT_CALCULATOR	1	10	0.50	2
P_POSITION	1	5	1.00	1
P_ATTITUDE	2	5	0.50	1
P_GUIDANCE	1	8	5.00	2
P_GUIDANCE_CTRL	1	6	1.00	1
P_SIMULATOR	1	2	1.00	1
P_MESSAGES	2	21	0.50	5
P_SENDING_CTRL	4	6	0.25	1
P_MMI	1	4	1.00	1
P_TELEMETRY	1	4	1.00	1
P_GNC	1	3	1.00	1
P_MMA	1	4	1.00	1

GNCL Object	DF	NP	COU	NT
UTILITIES	1	20	1.00	1
MATRIX	1	19	1.00	1
CMD_LINK	1	3	2.00	1
FB_LINK	1	3	2.00	1
INERTIAL_LINK	1	3	2.00	1
TM_LINK	1	3	2.00	1
E_BUS_1553	1	3	2.00	1
MATH_LIB	1	1	1.00	1
FLOATING_CHARACTERISTICS	1	2	1.00	1
LMMU	5	48	0.20	11
OP_CONTROL_START_LMMU	14	1	0.33	1
MMI	10	1	0.12	1
LMMU_PARAMS	1	7	1.00	1
LMMU_STATE	1	5	1.00	1
MMI_LIB	1	11	2.00	1
USER_CMD	1	4	1.00	1
DOWN_DATA_HANDLER	1	4	2.00	1
INERTIAL_SOURCE	3	4	0.66	1
G_RING	1	6	1.00	1
RANDOM_SRC	1	3	1.00	1
U_RAND	1	2	1.00	1
LAM	7	44	0.14	13
OP_CONTROL_START_LAM	5	1	0.50	1
LAM_LINK_HANDLER	4	1	0.25	1
ACQ_CONTROL	3	3	0.33	1
NAV_CONTROL	2	3	0.50	1
GUI_CONTROL	2	3	0.50	1
PIL_CONTROL	2	3	0.50	1
FSIM_CONTROL	3	3	0.33	1
TM_CONTROL	3	3	0.33	1
DISPATCHER	1	5	8.00	1
LAM_STATE	1	5	2.00	1
PROTECTED_AREA	1	14	6.00	3
G_AREA_GUARDIAN	2	8	0.50	1
SEMAPHOR	1	2	1.00	1
APROTECTOR	1	4	1.00	1

Table 2: *LOC and GNCL HOOD bjects metrics*

GNCO Object	DF	NP	COU	NT
ENVIRONMENTAL_SIMULATION	1	4	1.00	3
ACTUATOR_SIMULATION	1	2	1.00	1
DYNAMICAL_SIMULATION	1	1	1.00	1
SENSOR_SIMULATION	1	1	1.00	1
SIMULATE_SENSOR_SYSTEM	1	1	1.00	1
SW_MOCK_UP	1	30	1.00	10
INERTIAL_NAVIGATION	1	3	1.00	1
MISSION_CONTROL	1	27	1.00	9
MAN_MACHINE_INTERFACE	2	15	0.50	6
OUTPUT_HANDLER	1	1	1.00	1
CMD_BUFFER	1	3	1.00	1
INPUT_HANDLER	4	1	0.25	1
VT100_TERMINAL	1	6	2.00	1
EULER_BUFFER	1	2	1.00	1
KEPLER_BUFFER	1	2	1.00	1
GUIDANCE	1	10	1.00	2
ORBIT_TRANSFER	1	4	1.00	1
ORBIT_STANDARD	1	6	1.00	1
ORBITAL_FLIGHT_CONTROL	1	2	2.00	1

GNCR Object	DF	NP	COU	NT
P_TRACE	1	6	2.00	1
P_WORKLOADS	1	2	2.00	1
P_FLIGHT_PHASE	1	3	2.00	1
P_TIMER	3	5	1.00	1
P_MANAGER	10	2	0.20	1
P_GPA	3	29	0.66	6
P_SCHEDULER	7	7	0.50	1
P_FLIGHT_SOFTWARE	6	13	0.16	1
P_ACTUATORS	1	1	1.00	1
P_MMI_CONCENTRATOR	1	4	1.00	1
P_REPORT_CONCENTRATOR	1	1	1.00	1
P_SENSORS	1	3	1.00	1

Table 3: *GNCO and GNCR HOOD objects metrics*

3.2.3 Global Design Complexity Measurements

NT/NO: ratio between the total Number of Termir al objects and the total Number of Objects

That ratio allows to estimate the weight of global decomposition and the consistency of abstraction levels existing among pure logical objects and physical objects which will be given an implementing body.

The value of NT/NO ratio is at the most equal to 1, as shown by the dot filled area in the graph in figure 6.

The smaller NT/NO is, the more HOOD decomposition owns abstraction levels which are not justified.

A too functional decomposition in child objects does not contribute to the solution much more than the parent object does. A ratio too close to 1 could mean that a too physical decomposition with a great number of terminal objects was carried out.

The overall trend in application mock-up designs is that of going directly and rapidly to the point, infact, as it can be easily seen, we have something ranging from 0.92 to 0.68, that is we are dealing with quite physical

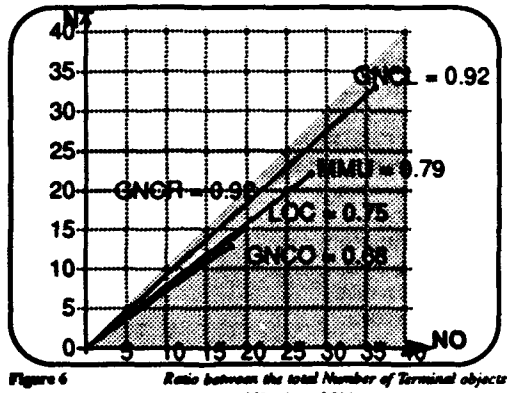


Figure 6 *Ratio between the total Number of Terminal objects and the total Number of Objects*

decomposition, too close to the implementation. The GNCL and GNCR, marking a ratio of 0.92, have a very-very physical decomposition. It seems that the only concern, leading the architecture design, was "how arranging an already fixed Ada implementation into HOOD objects"

NP/NT: ratio between the total Number of functions or Procedures PROVIDED and the total Number of Terminal objects

NP/NO: ratio between the total Number of functions or Procedures PROVIDED and the total Number of Objects

Figure 7 allows us to have a quick-look on the relevance of operations with regard to the objects. While the dark dotted histograms show the NP/NO ratio, on the other side, the light dotted ones give the NP/NT ratio. It could be useful recall now that NP takes into account the increasing of complexity of an interface due to the IMPLEMENTED BY mechanism, that is, NP is not merely the summation of the PROVIDED interfaces of the overall design.

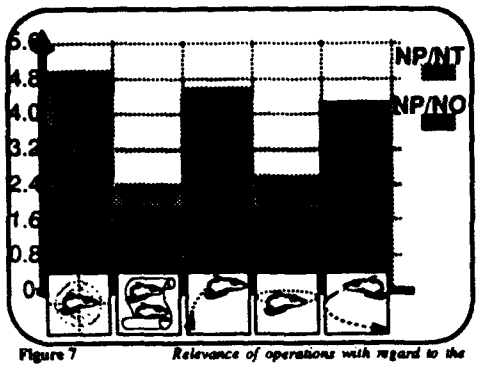


Figure 7 *Relevance of operations with regard to the objects*

3.3 Metrics of Specification Phase Complexity

This section is aimed at providing some metrics associated to the specification complexity, recalling in some way those already shown for the design complexity.

3.3.1 Model Structure Measurements

Figure 8 provides the total Number of boxes of the SADT model and the Maximum depth of functional decomposition.

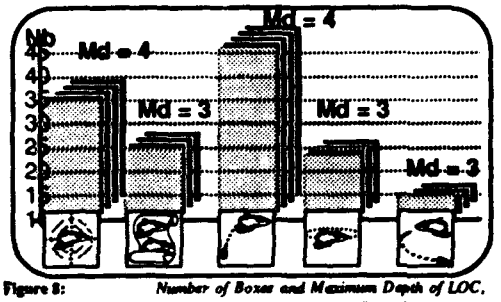


Figure 8: *Number of Boxes and Maximum Depth of LOC, MMU, GNCL, GNCO and GNCR SADT models*

Nb: Number of boxes
It should be considered only a dimensioning parameter, without any other direct complexity implication. It takes into account the diversity of all the activities owned by a model including also those activities due to the abstraction level mechanism (i.e. non-leaf boxes).

Md: Maximum depth
A decomposition level is added every time that an activity, estimated not enough specified, needs to be further split into child boxes. The problem statement complexity depends on that value, which only gives information about the deepest branch of the tree, without any other indication about the complete arborescence, being the depth of other branches unequal.
It is worth pointing out that the mock-ups SADT trees substantially mirror the corresponding design trees structure.

3.3.2 SADT Non-Leaf Boxes Metrics

In this section definition and meaning of metrics, which were calculated for each non-leaf box componing the SADT model, are given.

W: Weight
This metric measures the diversity of the basic activities included in a non-leaf box. The weight of a non-leaf box is the number of leaf boxes included in its descendants. The weight of a leaf box is conventionally set to 1.

Cc: Structural Complexity of communication
This metric measures the complexity of communication between the elements of the box decomposition. It is calculated as the sum of total number of connections between the box children and the number of interface points of child boxes attached to their parent.

Db: Dispersion of the breakdown
It is an indicator of the uniformity of the decomposition of a non-leaf module. It is equal to the standard deviation of the weights of the child boxes.

Cb: Cohesion of the breakdown
It measures the density of the links between the children of a non-leaf box. It is calculated as the ratio between the number of pairs of children, which are connected by at least one channel, and the total number of possible pairs of connected children.

Ci: Cohesion of the inheritance
It measures the density of the links between a non-leaf box and the elements of its decomposition. It is calculated as the ratio between the number of interfaces inherited by the child boxes and the number of child boxes of the non-leaf box.

Table 4 summarizes the SADT boxes metrics for each application software mock-up.

LOC Box	W	Cc	Db	Cb	Cl
MOCK-UP LOCALISATION	25.00	31.00	4.60	0.55	0.60
MANAGE	3.00	20.00	0.00	0.50	4.67
ACQUIRE	3.00	14.00	0.00	0.33	3.33
CALCULATE	14.00	12.00	0.00	0.00	6.00
DETERMINE ABSOLUTE ATTITUDE	7.00	15.00	1.50	1.00	4.50
READJUST IMUs ATTITUDE	5.00	27.00	0.00	0.25	3.00
EVALUATE ATTITTUDE	2.00	11.00	0.00	0.50	3.50
DETERMINE ABSOLUTE POSITION	7.00	21.00	0.83	0.50	2.25
PREDICT ORBIT	2.00	9.00	0.00	0.50	3.50
PROCESS FILTER	3.00	13.00	0.00	0.33	3.00
SEND	4.00	20.00	0.00	0.00	5.00

MMU Box	W	Cc	Db	Cb	Cl
ON BOARD MISSION SOFTWARE MOCK-UP	18.00	30.00	3.84	0.75	1.00
EXECUTE OPERATIONS PLAN	3.00	15.00	0.00	0.67	2.33
MANAGE VEHICLE SUBSYSTEM	11.00	19.00	1.09	0.00	4.75
MANAGE SUBSYSTEM	4.00	18.00	0.00	0.00	4.50
MANAGE PPT INTERFACE	3.00	10.00	0.00	0.33	2.00
MANAGE TM INTERFACE	3.00	11.00	0.00	0.33	2.33
MANAGE SYSTEM CONFIGURATION DATABASE	3.00	10.00	0.00	0.17	2.67

GNCL Box	W	Cc	Db	Cb	Cl
GNCL	32.00	50.00	3.09	0.50	1.67
ACQUIRE DATA	7.00	22.00	0.83	0.33	3.50
COMPENSATE & CONVERT IMUs DATA	3.00	10.00	0.00	0.33	2.00
CONSOLIDATE IMU DATA	2.00	8.00	0.00	0.50	3.00
PERFORM NAVIGATION	7.00	12.00	1.89	0.33	2.67
COMPUTE POSITION & VELOCITY	5.00	19.00	0.00	0.25	1.80
PERFORM GUIDANCE	5.00	12.00	0.50	0.00	6.00
COMPUTE ATTITUDE	2.00	7.00	0.00	0.00	3.50
COMPUTE REAL TRAJECTORY & ATTITUDE	3.00	13.00	0.00	0.33	3.00
PERFORM PILOTING	10.00	27.00	1.70	0.50	5.00
PERFORM PILOTING FOR CURRENT CYCLE	5.00	20.00	0.50	0.50	9.00
COMPUTE DEFLECTION & STATE VECTOR	3.00	16.00	0.00	0.33	4.00
COMPUTE ROLL COMMAND	2.00	18.00	0.00	0.00	9.00
PERFORM PILOTING FOR NEXT CYCLE	4.00	22.00	0.00	0.25	4.00
MANAGE FLIGHT SEQUENCE	2.00	9.00	0.00	0.50	3.50

GNCO Box	W	Cc	Db	Cb	Cl
GNC -ORBITAL PHASE MOCK-UP	17.00	24.00	0.80	0.50	0.40
HANDLE MAN-MACHINE INTERFACE	3.00	15.00	0.00	0.33	2.33
PERFORM ABSOLUTE NAVIGATION	5.00	16.00	0.94	0.33	1.33
PERFORM INERTIAL NAVIGATION	3.00	16.00	0.00	0.50	3.33
PERFORM GUIDANCE	3.00	7.00	0.00	0.17	1.67
PERFORM ORBITAL FLIGHT CONTROL	3.00	10.00	0.00	0.33	2.00
SIMULATE ORBITAL PH. ENVIRONMENT	3.00	10.00	0.00	0.33	2.00

GNCR Box	W	Cc	Db	Cb	Cl
MOCK-UP OF THE GPA IN ITS ENVIRONMENT	11.00	13.00	4.50	0.50	2.50
SIMULATE THE GPA	10.00	54.00	0.89	0.65	1.60
GUIDE THE SPACEPLANE DURING ATM.PH.	3.00	31.00	0.00	0.83	6.33
PILOT SPACEPLANE DURING ATM.PH.	3.00	21.00	0.00	0.33	5.67
STABILIZE SPACEPLANE DURING ATM.PH.	2.00	13.00	0.00	0.50	5.50

Table 4: *GNCL, GNCO and GNCR SADT boxes metrics*

3.3.3 Global Specification Complexity Measurements

W/Nb: ratio between the Weight of the model and the total Number of boxes
That ratio allows to estimate the relevance of abstraction levels with regard to the global decomposition.
The value of W/Nb ratio is at the most equal to 1 (only one decomposition level), as shown by the dot filled area in the graph in figure 9.

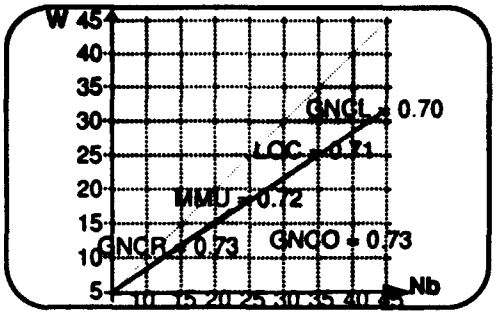


Figure 9: *Ratio between the Weight and the total Number of Boxes*

As it can be easily seen there is no significant difference in how the problem statement was specified using SADT methodology.

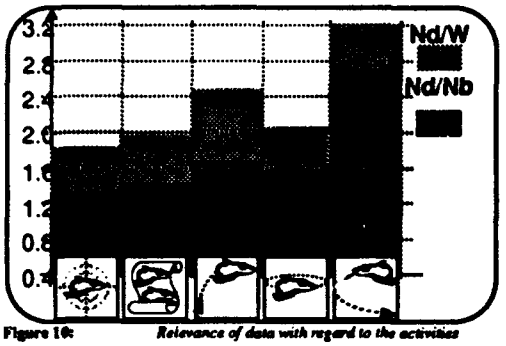


Figure 10: *Relevance of data with regard to the activities*

Nd/W: ratio between the total Number of data and the model Weight

Nd/Nb: ratio between the total Number of data and the total Number of boxes

Figure 10 allows us to have a quick-look on the relevance of data with regard to the activities.
While the dark dotted histograms show the Nd/Nb ratio, on the other side, the light dotted ones give the Nd/W ratio. It could be useful recall now that Nd is an indicator of the information quantity present in the model.

4. MOCK-UP PROJECT TECHNICAL SURVEY

Giving a feeling of experienced methods, being supported by the calculated metrics, without any pretence to a rigorous approach or an accurate evaluation, is just the main goal of this synthesis.

- As qualitative conclusion, it seems that:
- the problem specification was approached more regularly and homogeneously than the solution definition was;
- all the designs were approached mainly with the concern of fitting the HOOD architecture into an already existing Ada implementation as problem solution;
- the HOOD hierarchical abstraction mechanism was felt more as an obstacle, rather than a[n] efficient way of organizing the software implementation.

Finally, it is worth noticing that the values of ratios

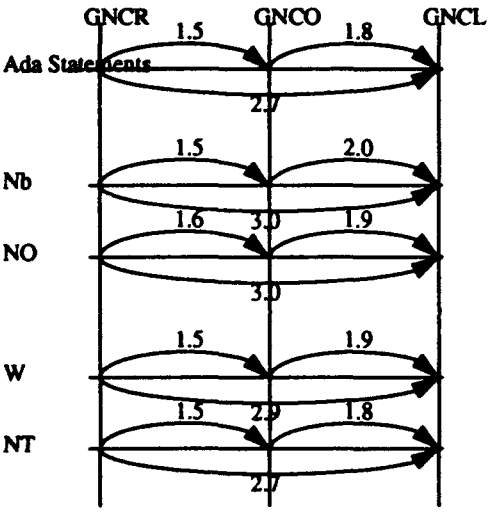


Figure 11: *Ratios between code size and SADT-HOOD dimensioning parameters*

between the code size of GNC application mock-ups are quite similar to those between the corresponding SADT-HOOD dimensioning parameters (see figure 11).

5. CONCLUSION

The main results we got in the Mock-Up experience are the following:

1. applicability of methods for the On-Board software development

The software engineering methods: SADT, HOOD and the ADA language were successfully used in the Mock-Up development process.
It means that these methods:

- can be used together in the same development process; SADT for the requirement phase, HOOD for the design and Ada language for the coding phase;
- can be used in a distributed environment involving several companies on several geographical sites;
- generally improve the software development productivity and quality, even if some aspects are not covered (e.g. real time).

All these evaluation elements are obviously only qualitative and do not preclude the possibility of using other methods. Anyway, we feel that, from a management point of view, such methods allow a clear understanding and a good visibility of the software design and thus a good control of the software development.

2. methods mastered by the space software industry
   All the companies involved in the HERMES On-Board software development successfully used these methods. It means that the methods are mature enough to be used in an industrial development, it also means that the industrials are trained and aware of the use of these methods.
   It is also worth pointing out that the use of common methods allows a standardization and an harmonization inside a software project, every participant speaks with the same language and has the same understanding.
3. availability of metrics
   The main pragmatic result of this study is the availability of statistics which allow to define some metrics or ratio linking:
   - the complexity of the problem (according to SADT metrics)
   - the complexity of the design (HOOD metrics)
   - the size of the SADT model: number of boxes and arrows
   - the size of the design: number of objects, number of operations
   - the software size: number of lines of code.

   These ratio are available for:
   - management purposes, as estimation figures to evaluate with more accuracy the software development effort;
   - development control, to verify the adequation of the design effort to the problem complexity;
   - quality assurance, these ratio are also very useful to check if the software complexity is globally under control.

Discussion

Question C. KRUEGER

What was your experience with using HOOD in your project relative to the real-time performance of your system? Does HOOD provide the features needed to support real-time performance?

Reply

Real-time performances should be measured on the target system. Unfortunately, our mock-up development was designed to be executed on a "host" computer. Real-time performances of HOOD in providing support for real-time architecture has been pointed out through this experience.

CONCEPTION LOGICIELLE DE SYSTEME REACTIF AVEC UNE METHODOLOGIE D'AUTOMATES UTILISANT LE LANGAGE LDS.

JJ. Bosc
Thomson-CSF, division CNI
46/47, Quai le Gallo
92103 - Boulogne-Billancourt
France

SOMMAIRE :

La phase de conception d'un système réactif est certainement la plus délicate, ne serait-ce que parce que les choix qui seront fait ne pourront être confirmés qu'en phase d'intégration du logiciel avec le matériel. Car c'est seulement à ce moment là que le respect des contraintes temps réel pourra être vérifié.

Il faut donc adopter une méthode rigoureuse prenant en compte les spécificités du problème à résoudre, supportée par un formalisme, le langage LDS. Jusqu'où peut-on aller avec ce langage, et quelles sont les techniques de codage associées ?

INTRODUCTION :

Après avoir exposé les besoins d'un formalisme pour la conception, qui ressortent après la phase d'analyse d'un système réactif, nous verrons la possibilité et les solutions proposées autour du langage LDS.

Un système réactif est caractérisé par :

des contraintes temps réel;

plusieurs processus asynchrones qui sont en général des machines à états;

un flot de contrôle important : combinatoire importante d'états et d'événement ;

peu de transformation de données.

Ce type de système se rencontre fréquemment dans le domaine des applications de télécommunication par exemple.

Les spécifications logiciel de tel système, et de beaucoup d'autre, sont en général de nature fonctionnelle, et répondent à la question : "que faut-il faire ?"
Mais la prépondérance du flot de contrôle, et les contraintes temps réel conduisent à analyser et à spécifier également le comportement dynamique.

Ainsi l'analyse débouche typiquement sur trois types de description :

- description statique ou fonctionnelle : qui permet d'exprimer que la fonction d'un système complexe se décompose en sous-fonctions, qui elles-mêmes peuvent être décomposées et ce jusqu'à identifier des fonctions élémentaires. Classiquement une telle description peut s'appuyer sur la méthode SA (Analyse Structurée). Chaque niveau de décomposition peut être décrit par un diagramme de flot de données qui précise quelles sont les informations échangées par les sous-fonctions représentées par des processus de transformation de données.

- Description du comportement : qui exprime la manière d'activer les processus de transformation. Chaque configuration de processus actifs simultanément à un instant donné, représente un mode de fonctionnement du système. Chaque changement de mode de fonctionnement et les conditions du changement peuvent être décrits par un diagramme de flot de contrôle.

- Description dynamique : permet de spécifier la logique et le séquencement des échanges d'informations entre processus simultanément actifs, correspondant à un mode de fonctionnement, et le monde extérieur.
Dans un mode de fonctionnement donné, tous les processus actifs ne participent pas forcément à un même type d'échange. Il peut être intéressant de ne faire figurer dans un scénario que les processus acteurs. Ainsi à un mode de fonctionnement peuvent être associé plusieurs scénarios.
Les scénarios sont eux-même raffinables, et peuvent être décomposés en sous-scénarios à chaque niveau d'abstraction auquel correspond un diagramme de flot de données.
Ces scénarios peuvent être décrits graphiquement à partir de diagrammes d'échanges où les exigences temps réel peuvent être indiquées, et permettent de spécifier l'ensemble des protocoles gérés par le système.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

REPONSE AUX BESOINS DE CONCEPTION :

Les performances temps réel de la conception sont essentiellement liées aux parallélisme des processus. Il est donc important d'identifier très tôt le parallélisme, permettant de dégager des processus communicants qui seront autant de machines à états.

Il est tentant d'adopter une démarche orientée objet pour des raisons évidentes de maintenabilité, de réutilisabilité, ...
Mais là encore la prise en compte du flot de contrôle, et le respect des trois analyses fonctionnelle mais surtout comportementale et dynamique, conduit à s'intéresser d'abord aux événements à traiter.
Les deux approches ne sont d'ailleurs pas incompatibles : "Un objet a un état, un

comportement et une identité" (G. BOOCH). La notion d'état est le critère numéro un pour identifier les objets actifs, qui devront traiter l'ensemble des événements en provenance du monde extérieur, et qui devront réagir par rapport eux.

Le premier niveau de l'analyse fonctionnelle identifie un certain nombre de processus traitant des informations en provenance directe du monde extérieur. C'est l'environnement qui impose le parallélisme nécessaire de l'application, ainsi les processus figurant à ce niveau fournissent les grand axes du parallélisme à identifier.

En se laissant orienter par le flot de contrôle, les fonctions identifiées dans chacun des diagrammes de flot de données doivent être regroupées de manière à autoriser la simultanéité des scénarios définis dans chacun des modes de fonctionnement de l'application. Ces regroupements permettent d'identifier les objets actifs représentant le parallélisme strictement nécessaire.

Un certain nombre de conflits peuvent apparaître à l'occasion de ces regroupements liés à l'accès aux données partagées rémanentes. Afin d'éviter l'apparition de données globales, toujours dangereuses dans les systèmes multitâches monoprocesseur, l'encapsulation est souhaitable.
L'encapsulation permettant de mettre en place l'exclusion mutuelle peut conduire à faire apparaître de nouveaux processus. L'ensemble des processus pourront accéder aux données rémanentes par messagerie avec ces processus nouvellement identifiés, ce qui néanmoins peut être coûteuxen temps.

Le formalisme nécessaire durant toute cette démarche critique d'analyse doit répondre aux besoins suivants :

- approche hiérarchique structurée : permettant de jalonner les différentes étapes de regroupement, conduisant aux objets actifs.
- La modularité : identifications des objets actifs qui encapsulent les données, et de leurs interfaces.
- Concepts de base du temps réel : parallélisme, communication, gestion du temps, ...
- L'expression du comportement des objets actifs sous forme de machines à états.

L'utilisation sans précaution de la programmation structurée pour exprimer le comportement conduit dans bien des cas à une mauvaise lisibilité et maintenabilité du logiciel. Les raisons principales sont les suivantes :

- les traitements du flot de contrôle et du flot de données sont mélangés, et il est très difficile de distinguer les variables d'états et les événements des données.
- Un grand nombre de variables d'état de type simple, exemple un booléen, tout au long du programme provoque fréquemment un nombre important de tests de ces variables, afin de déterminer l'action à produire. Ceci a pour effet d'obtenir très vite localement un nombre d'imbrications inacceptable, ou d'engendrer des tests complexes et multiples, ce qui dans les deux cas nuit à la lisibilité.
- La structure de contrôle, qui devrait avoir en charge la mise à jour des variables d'état, ne peut pas être facilement isolée. De ce fait elle se retrouve diluée dans l'application, et variables d'état et événements sont consultés et modifiés de partout (variables globales), ce qui dans un contexte multitâche est particulièrement dangereux.
- Lorsque les états n'ont pas été clairement identifiés au préalable, le risque est grand de voir apparaître des variables d'état au fur et à mesure de l'élaboration de la conception détaillée. Le foisonnement des variables d'état a pour conséquence de complexifier inutilement le contexte ce qui rend le logiciel difficilement testable.

L'analyse du comportement devient délicate, la description peut difficilement être exhaustive, la prise en compte de nouveaux états peut entraîner des effets de bords et des régressions.

Il faut donc se soucier en phase de conception de limiter le nombre de variables d'états en les regroupant.

On peut remplacer n variables binaires par une seule variable pouvant avoir 2^n valeurs

au plus, car parmi toutes ces valeurs toutes ne sont pas forcément pertinentes.

Il est en général plus simple dès la conception d'identifier la liste des "états" possibles pour chacun des processus.
Il vient ainsi plus naturellement de positionner une seule variable pour gérer chacune de ces listes.

LE LANGAGE LDS :

Un formalisme répond aux besoins exposés ci-dessus, et a été spécialement défini pour l'analyse et la description des applications de télécommunication par le CCITT (recommandations Z.100) : le langage SDL (Spécification and Description Language, LDS en français).

Ce langage propose une double syntaxe, graphique et textuelle, ce qui fournit une bonne lisibilité du modèle et permet la vérification, la simulation et la génération de code.

Il existe trois types de description en LDS correspondant à trois étapes de conception :

- description hiérarchique
- description des moyens de communication
- description du comportement

L'architecture de l'application est décrite à l'aide du premier type de description. La racine de "l'arbre" représente l'ensemble du "système" qui peut se décomposer en "blocs". Chaque "bloc" pouvant à son tour être constitué de "blocs", et ce autant de fois que souhaité, ou en "processus".

La sémantique associée au "bloc" est assez libre. Un "bloc" peut représenter une famille de fonctions, une sous-application résidant sur un processeur, ou plus simplement un certain niveau d'abstraction.

Un "processus" représente une machine à état, qui peut être décrite par un automate unique ou constitué de plusieurs automates : les "services". Les "processus" ou "service" peuvent faire appel à des sous-automates appelés "procédures".

A chaque nœud de l'arbre on peut associer des déclarations textuelles (déclaration de type, de "signaux",...). Les variables ne peuvent être déclarées que sous un "processus".

A chaque niveau ("bloc", "processus", service") les entités identifiées dans la vue hiérarchique peuvent être "reliées" entre-elles par des "channels" ou des "routes". Chaque lien synthétise un sous-protocole, et identifie la liste des "signaux" qui peuvent être échangés.

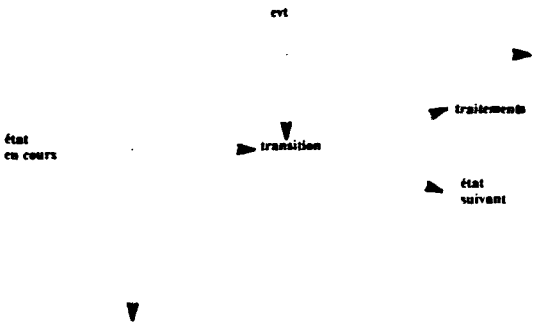
C'est le deuxième type de description qui permet de représenter ces liaisons.

Enfin le troisième type de description est utilisé pour décrire le comportement des "processus", des "services", et des "procédures", sous forme d'automates à états finis.

Si on appelle "état" une combinaison particulière des variables de l'application dans une situation donnée et "événement" toute information nouvelle susceptible de modifier le contexte, un automate d'états finis décrira toutes les combinaisons "états/événements" possibles.

L'automate peut être représenté sous forme d'une matrice avec les états en ordonnée et les événements en abscisse. Chaque couple état/événement représente une situation envisageable.

Dans un état donné, l'occurrence d'un événement entraîne l'exécution d'une "transition". Celle-ci consiste à effectuer un ou plusieurs traitements et se termine par un changement d'état.



Le contexte n'est pas forcément représenté complètement par "l'état". Il peut être utile de conserver des variables d'état définissant des états secondaires. D'autre part, le résultat d'un traitement doit pouvoir influer sur l'état suivant en fin de transition. Ainsi le traitement d'un événement peut déclencher des conditions de transitions vers plusieurs états suivants possibles.

Les avantages d'une description sous forme d'automates sont :

- La prise en compte d'une combinatoire états/événements importante : cette approche pousse à se poser dans chaque état toute les bonnes questions, et à prévoir également les cas présumés "impossibles" (traitements d'exception), qui conduit à une description exhaustive.
- La lisibilité : les automates se déduisent naturellement des protocoles à implémenter.
- La traçabilité : le contexte est simple, c'est l'état de l'automate, ce qui garantie une bonne testabilité du logiciel.

La limitation du code procédural : réservé aux traitements des données.

Le LDS offre des mécanismes, spécifiques du temps réel, accessibles à l'intérieur de toute transition :

création/destruction d'instance de processus;

communication entre objets;

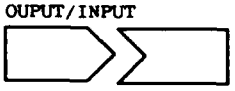
gestion du temps.

Le nombre d'instances de processus peut varier au cours du temps (création/destruction). Chaque instance est identifiée à l'aide de son PID (Processus IDentifier).

Les communications entre objets sont de trois types :

communications de type asynchrone, réservée aux communications entre processus :

OUPUT/INPUT



Chaque processus possède une file d'entrée de type "fifo" destinée à réceptionner les événements en provenance d'autres processus.

Communications de type synchrone :

PRIORITY OUPUT/INPUT

(réservées aux communications entre services d'un même processus)

SAVE

(sauvegarde des événements externes que l'on ne souhaite pas traiter dans l'état en cours.

- Appels procéduraux :

CALL

(appel d'un sous-automate).

TASK

(appel d'une fonction déclarée dans un type abstrait).

La notion de temps en LDS intervient à travers deux mécanismes :

- activation d'une temporisation : sur échéance un événement est produit dans la file du processus.
- désactivation d'une temporisation : la temporisation à désactiver est identifiée par l'événement à produire.

Le langage LDS propose de raisonner d'emblée en terme d'automate, et l'analyse se trouve de ce fait guidée. Mais ce serait une erreur de vouloir tout représenter en LDS.

En effet le langage LDS traite de manière imparfaite le problème de représentation des données. Il n'existe pas de notion de pointeur, et le langage ne répond pas au besoin de description de structures de données complexes ce qui entraîne une description LDS qui ne correspond plus au code.

D'autre part la description manque de concision, il faut donc réserver l'utilisation du LDS pour des descriptions "macroscopiques" du fonctionnement.

Il est clair que le LDS est mal adapté à la description des données et donc des traitements qui s'appliquent à ces données. Toutes les tentatives en ce sens débouchent sur un constat d'échec.

Il est donc très important de savoir s'arrêter dans la description détaillée. La difficulté est de déterminer la frontière ou la limite d'utilisation, d'autant que le LDS apparaît comme trop riche syntaxiquement (110 mots clés dans la norme 88) et à la fois trop pauvre sémantiquement, notamment par rapport aux langages évolués de programmation tel C ou Ada.

L'utilisation du LDS suppose qu'au préalable ces limites et les restrictions d'utilisation soient clairement définies avant de débuter la phase de conception. Au minimum ces règles doivent indiquer que les traitements sont encapsulés dans des types abstraits (NEWTYPE) ou bien encore décrits de manière informelle (TASK 'mettre à jour la table des abonnés').

L'utilisation du LDS en conception suppose l'utilisation d'un exécutif temps réel pour implémenter les différentes notions proposées par le langage.

Il est important de définir au préalable les règles d'utilisation du LDS et les restrictions syntaxiques et sémantiques : mots clés autorisés et choix d'implémentation. La conception peut en effet être très influencée par ces choix.

Ces difficultés maîtrisées, les points forts du LDS sont :

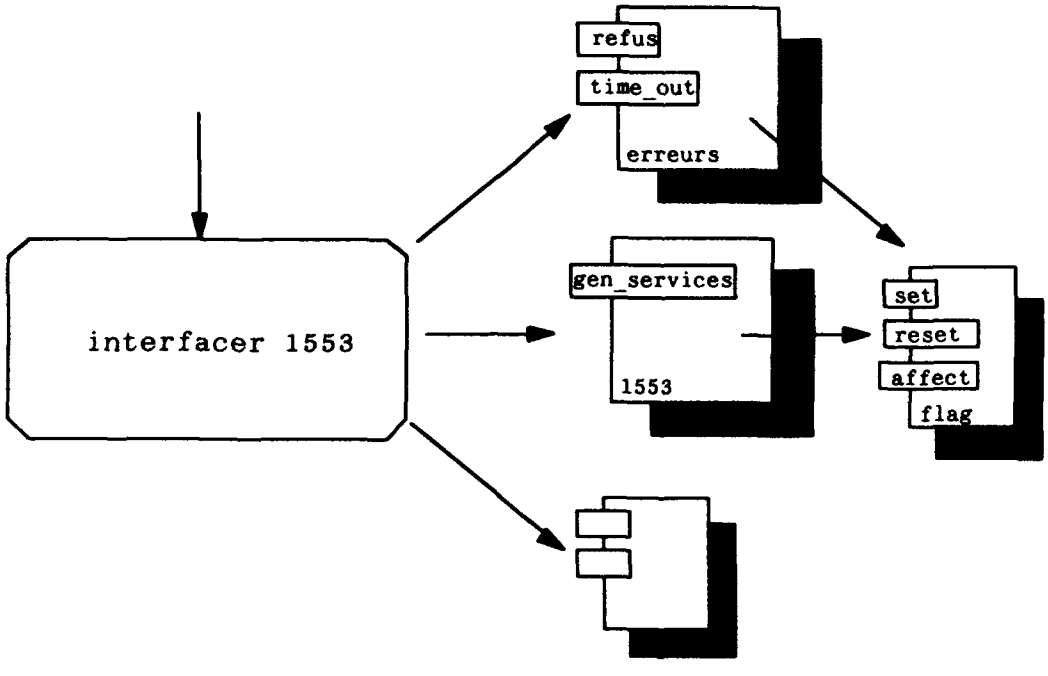
- la description de l'architecture logicielle,
- le travail en équipe facilité,
- un guide à l'analyse et à la réflexion lors de la description des automates.

DEMARCHE DE CONCEPTION EN LDS :

Les différentes étapes de la conception préliminaire sont :

- déterminer les interfaces avec l'extérieur.
- Identifier les événements correspondants. En pratique un événement par type d'information échangée.
- Identifier les processus (objets actifs).
- Description de l'architecture de l'application à l'aide des "blocs" et des processus.
- Déterminer pour chaque processus l'interface constituée de l'ensemble des événements ("signal" en LDS) entrant dans le processus.
- Identifier les ressources manipulées par chaque processus.
- Identifier les modes de fonctionnement (états) pour chaque processus.

Les traitements des données peuvent être abordés hors LDS, en utilisant les types abstraits qui permettent l'abstraction des données.



données abstraites

Ainsi à chaque processus peut être associée une hiérarchie de types abstraits de données. Chaque type abstrait fournit la liste des opérateurs ou fonctions qui peuvent s'appliquer aux ressources rémanentes qu'il encapsule.

L'objectif de la conception détaillée est de décrire le comportement de chaque processus conformément aux modes de fonctionnement identifiés dans la phase précédente.

Ici deux cas de figures peuvent se présenter :

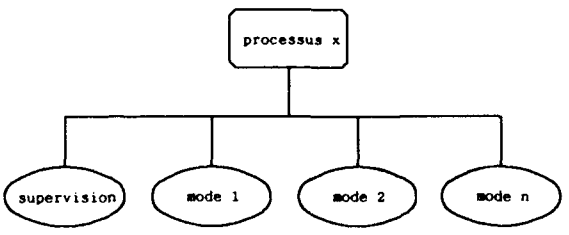
- Les modes de fonctionnement sont peu nombreux (moins de 20) et ils correspondent directement aux états de l'automate. La description de l'automate est simple et représente complètement le comportement du processus.

- Les modes de fonctionnements sont complexes, c'est-à-dire qu'ils conduisent à une décomposition de chacun d'entres-eux en plusieurs états.

  Si l'on souhaite minimiser le nombre de variables d'état, l'automate correspondant peut très vite devenir difficile à maîtriser car la combinatoire états/événements diverge rapidement.
  On peut dans ce cas utiliser la notion de "service" du LDS.

  Chaque service peut être associé à une famille d'états, ou à un mode de fonctionnement. Ce n'est pas simplement dans ce cas, une découpe modulaire de l'automate, en effet l'état du processus devient alors la combinaison de tous les états des services le constituant, ce qui permet de réduire sensiblement le nombre total d'états à décrire.

  Si l'on adopte cette démarche il peut être utile de confier à un des services le rôle de chef d'orchestre ou de supervision. Les états de ce service synthétisent les différentes combinaisons des états des autres services, ils représentent des "super-états" correspondant alors aux différents modes de fonctionnement du processus.



Lorsqu'un événement externe au processus doit être traité dans tous les modes de fonctionnement, il doit alors être traité par le service de supervision, ce qui permet de factoriser les transitions associées.

Il peut être utile de confier au service de supervision le soin de réceptionner tous les événements externes, afin de traiter au plus tôt les changements de mode. Le service de supervision dans ce cas se charge de sous-traiter aux services concernés les traitements associés aux événements, ou simplement de les prévenir d'un changement de mode (synchronisation des automates).

Ces communications entre services, internes au processus, doivent clairement être identifiées.

Les différentes étapes de la conception détaillée sont :

décomposition des processus en services (si nécessaire);

répartition des événements externes par rapport aux services;

- identification des communications, et de l'interface de chaque service;
- identification des états pour chaque service;
- identification des traitements utilisés par chaque service;
- description du comportement : automates à états finis;
- description des traitements : programmation structurée.

CODAGE ET GENERATION DE CODE :

Arrivé à ce stade, l'ensemble du comportement du système est décrit de manière très complète, et la plupart des traitements des données sont identifiés.

Il est dès lors tentant d'utiliser une génération de code automatique des automates à partir de la forme textuelle du LDS.

Le code généré s'appuie obligatoirement sur un noyau temps réel, et le code exécutable généré peut utiliser directement les primitives du noyau.

Une autre approche consiste à générer des tables interprétables, correspondant aux matrices états/événements dans lesquelles est indiquée pour chaque couple états /événements (ou transition) la liste des traitements à exécuter.

Cette solution permet d'isoler la structure de contrôle de l'application, qui devient un interpréteur de tables. Cette structure de contrôle peut ainsi être facilement optimisée en encombrement mémoire et en rapidité. Il est souhaitable que l'interpréteur de table intègre des fonctions de traçage des objets LDS manipulés, qui seront très utiles en phase de mise au point du logiciel, ou pour mettre en place une politique de test systématique. En effet les informations ainsi tracées sont relatives au modèle de conception et permettent de vérifier le comportement de l'application.

Cette structure de contrôle joue le rôle d'interface d'appel entre l'application et le noyau temps réel, ce qui assure une certaine indépendance par rapport à ce dernier. Le générateur de code et la structure de contrôle sont donc étroitement liés et doivent être définis simultanément.

D'autre part il peut être très utile que le générateur de code fournisse les interfaces avec les traitements de données qui eux seront codés à l'aide d'un langage de programmation classique. La génération des déclarations et des "squelettes" des procédures ou fonctions peut être envisagée.

L'avantage d'une telle solution réside avant tout dans l'obtention d'un code conforme à la conception de l'application. L'expérience montre que la moitié du code total à produire peut ainsi être déduit automatiquement à partir de la conception.

L'analyse d'un système réactif débouchant sur trois descriptions fonctionnelle, comportementale et dynamique, puis l'utilisation du formalisme LDS en phase de conception en respectant un certain nombre de règles, et finalement une génération de code automatique des automates, fournissent les moyens de mettre en place un processus de production cohérent de bout en bout.

Discussion

Question Mr JANVIER

Avez-vous ressenti le besoin de valider formellement les automates au niveau spécifications, avant de passer à la réalisation?

Reply

Ce besoin n'a pas été ressenti sur les projets actuels car ils sont de taille moyenne et le comportement du système est bien maîtrisé par les spécialistes du domaine. Cependant, les fonctions des systèmes réactifs de communication allant en se complexifiant, cette validation formelle deviendra certainement nécessaire dans l'avenir.

Question G. LADIER

Cette méthode facilite-t-elle ou nuit-elle à la réutilisation?

Reply

Au niveau conception, la modification d'un automate est d'autant plus difficile que la décomposition en états est transformée. De ce point de vue, la réutilisation doit être limitée à de faibles évolutions.

Artificial Intelligence Technology Program at Rome Laboratory

Robert N. Ruberti
and
Louis J Hoebel

Knowledge Engineering Branch
Rome Laboratory
Griffiss AFB USA 13441-4505

Abstract
This paper provides an overview of the Artificial Intelligence program at Rome Laboratory.The three major thrusts of the program are described. The Knowledge-based planning program seeks to develop the next generation of AI planning and scheduling tools. The engineering of knowledge-based systems focuses on the development and demonstration of technology to support large-scale, real-time systems of knowledge-based components. The knowledge-based software assistant program seeks to develop a new programming paradigm in which the full life-cycle of software activities are machine mediated.

1. Introduction

Rome Laboratory (RL), an Air Force laboratory, focuses on the development of Command, Control, Communications and Intelligence(C3I) and related technological capabilities for the Air Force. RL is designated as a *Center of Excellence in Artificial* Intelligence based on its extremely successful track record of research over the past decade.

The goal of Rome Lab's Artificial Intelligence(AI) program is to develop the technology in Air Force needs areas and demonstrate applications to C3I problems. The program's scope ranges from research and development extending the intelligent functional capabilities of AI technology, to generic tools and methods in broad areas of interest, to the use of AI in application specific programs. Application programs are addressed by five different Rome Lab Directorates based on their separate mission responsibilities. These programs include applications in survivable adaptive planning, intelligence indications and warning, smart built-in-tests for electronic components, tactical command and control, communications network control, adaptive surveillance and conformal antennas. The technology base program addressing component level technology and generic tools is described in the remainder of this article.

2. The Technology Base Program

Although state-of-the-art AI is sufficiently mature for many near term applications, there are critical technology shortfalls to address before the breadth of potential applications envisioned can be realized. The technical opportunities for the Air Force include a wide variety of decision support systems which are overwhelmed by information, response option complexities and response time requirements. With built-in intelligence these systems can overcome and improve their performance where it is currently limited by conventional programming approaches. Therefore, the technology base program has been structured to address the generic technology needs common to these applications. These needs areas include real-time AI, parallelism in AI, distributed and cooperative problem-solving, AI acquisition and development methodologies, intelligent man-machine interaction, explanation in expert systems, knowledge base maintenance, reasoning with uncertainty, knowledge engineering for large scale systems and verification and validation techniques. To provide additional focus technology is being advanced in three major thrust areas. Knowledge based planning, knowledge based systems engineering and knowledge based software assistance.

2.1 Knowledge Based Planning

The objective of this program thrust is to support the rapid, accurate and efficient creation and modification of plans: sequences of action and events designed to achieve certain goal conditions or states in various operational environments. There have been developed a series of technology feasibility demonstrations in the domain of tactical mission planning that have led to operational prototype systems. The primary applications for this technology range from conventional robotics planning associated

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

with on-board satellite control to planning of resource allocation in tactical or strategic mission planning, to planning of a "trajectory" a piece of material might take as its path from point of manufacture to point of consumption in logistics. Planning approaches differ in the degree to which there is a man in-the-loop of the planning process, the degree to which the plans are unique or stereotypical, the rate at which changes occur in either the environment, the plan or the goal structure upon which the plan is based, as well as in the temporal, causal, resource and task complexities of the plan.

A new initiative focuses development activities in this thrust on the next generation of generic planning, resource allocation, and scheduling technology to achieve an order of magnitude performance enhancement over current operational planning systems. The transportation planning and scheduling requirements associated with force deployment in direct support of world-wide force projection goals are being addressed. AI planning techniques are being developed to meet these daily planning activities of operational commands. The principal product will be an integrated, well-engineered and validated suite of AI planning tools ready for application to this and other operational planning domains. Opportunities for technology advancement exist in the areas of opportunistic reasoning about resource contention, planning in the large, intelligent reuse of plans, integration of AI planning and decision analysis, real-time situated planning, plan-based situation assessment and distributed planning. As a joint Rome Lab and Defense Research Projects Agency (DARPA) initiative, Rome Lab's responsibilities are to identify and address shortfalls in projected operational capabilities based upon current planning and scheduling technology, and to pursue through feasibility demonstration the development of new technology solutions.

An early success story under this Initiative was the Dynamic Analysis and Replanning Tool (DART), developed on site at USTRANSCOM to meet specific requirements for Operation Desert Shield.

As U.S. military forces pull back and contract to a "Fortress America" posture, the importance of strategic mobility will dramatically increase, since the luxury of forward positioning will be gone. The demanding conditions surrounding Operation Desert Shield/Storm emphasized the importance of timely crisis action planning in dealing with rapid, massive deployments of force.

The goal of the DARPA/Rome Lab initiative is to develop and demonstrate the next generation of generic Artificial Intelligence (AI) planning, resource allocation, and scheduling technology focused on achieving significant performance enhancements over current DoD operational planning systems. The principal product stemming from this investment will be an operationally validated suite of integrated planning tools that will address the large scale planning, analysis, and replanning problems typified by strategic deployment planning.

These tools will help the CINC and his staff evaluate proposed courses of action. The system would allow the rapid application of qualitative criteria or decision rules to a variety of planning scenarios, and facilitate rapid response to unforeseen changes in plan assumptions, outcomes of actions, or external conditions. The tools will also aid in the generation and maintenance of the force and deployment databases.

The operational focus of this initiative is transportation planning and scheduling associated with force deployment, specifically deliberate planning and crisis action planning tasks at the National Command Authority, the Joint Staff, the major Unified Commands, and the US Transportation Command (USTRANSCOM) and its service components: Military Airlift Command (MAC), Military Sealift Command (MSC), and Military Transportation Management Command (MTMC).

This initiative is divided into three closely-coordinated tiers of activity. Tier 1 is the generic technology development and demonstration tier where shortfalls in projected operational capabilities based upon current planning and scheduling technology will be identified and

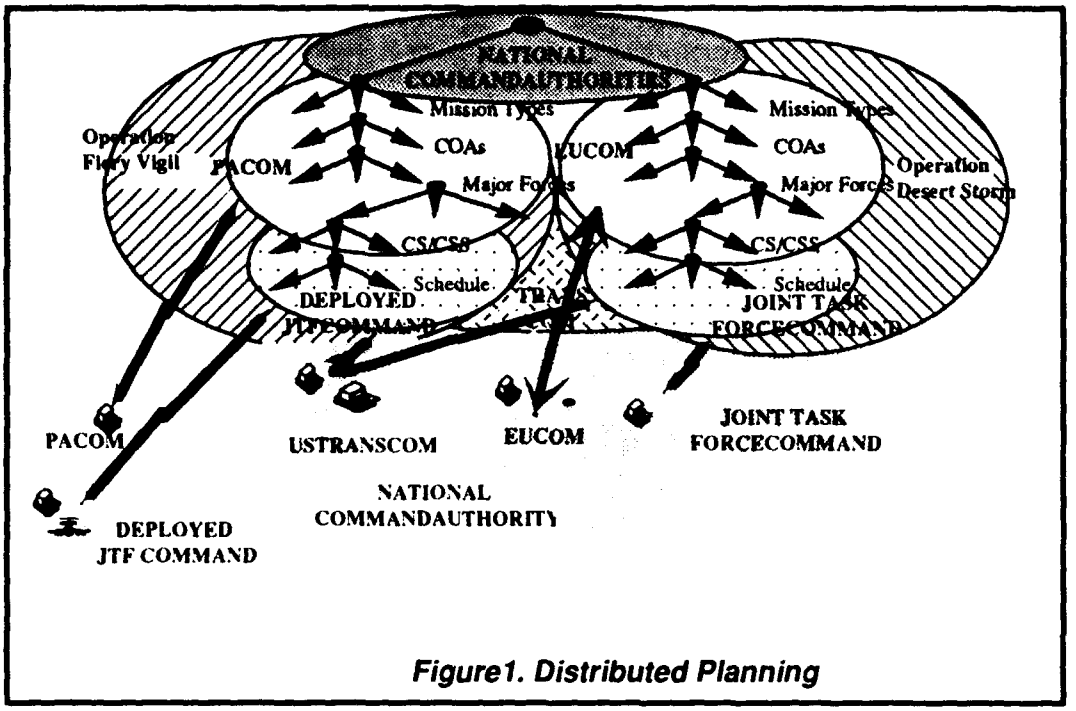


Figure1. Distributed Planning

addressed. In Tier 2, promising technology solutions will be integrated into technology feasibility demonstrations targeted at specific parts of the crisis action planning problem. In Tier 3 operational prototypes based on mature components and technical successes of Tier 2 will be developed and fielded through integration into the ongoing modernization programs of specific user communities.

The interface between these three tiers will be bridged by a common prototyping environment which will provide a ready medium for two-way flow of technology and domain knowledge between the research, application, and operational communities. This prototyping environment, including hardware, software, planning and scheduling tools, and domain faithful suites of test data and intelligent simulations, will be available to all participants in either tier on a "mix and match" basis. This prototyping environment will not only serve as the initiative vehicle for technology evaluation and transfer, but will evolve into an architecture for advanced C4 systems. Maturing research products that have been thoroughly tested in the environment will transition into operational prototypes. The successful development of the Dynamic Analysis and Replanning Tool (DART), in just ten weeks, was the first application for this process.

DART was developed to solve the deployment force resequencing problem. Initial technology prototyping experiments were conducted from March to August 1990. Late in August, USTRANSCOM requested Initiative help in accelerating the development due to Desert Shield requirements. In 10 weeks, the DART system was developed with DARPA funds and Rome Laboratory technical support.

The system uses a relational data base, graphical editing tools, and closely coupled simulation to speed modification of TPFDDs (Time Phased Force and Deployment Databases) and analysis of operational plans. DART resides on a Sun workstation and can exchange data with WWMCCS hosts. An open system architecture was a design requirement and a large portion of the DART software is commercial off-the-shelf. A second phase is currently under way to enhance and productize DART. While DART provides some plan handling and analysis capability, initial force generation planning remains a manual, error-prone process.

USTRANSCOM used initial prototypes to make deployment decisions early on in Desert Shield. During October, DART was used and positively impacted analysis conducted by USTRANSCOM. The resulting deployment was the largest ever in the associated time period. In November, DART was demonstrated to CINCTRANS and immediately fielded to Europe to assist CINCEUR in deploying tanks and personnel to

Saudi Arabia. CINCEUR planners were able to use DART after a single day of training. USTRANSCOM planners have also used DART as a key tool for redeployment planning of troops and material back from the Persian Gulf theater. DART has been qualitatively compared to the JOPES on WWMCCS. The system facilitated an order of magnitude speed-up of the editing and analysis cycle used by the Crisis Action Team at USTRANSCOM. The graphics in DART also improve upon the JOPES interface, enhancing the ability to visualize plans and smoothing the learning curve considerably.

DART has shown that technology can rapidly respond to the needs of deployment resequencing. The Rome Laboratory/DARPA Planning Initiative will address the entire spectrum of Strategic Deployment Planning requirements. Current work in distributed planning, as represented in Fig. 1 above, will culminate in an integrated feasibility demonstration. This demonstration will support concurrent planning at physical distributed sites at Hawaii, St. Louis, Rome and Washington, DC. The demonstration scenario will support simultaneous activities such as occurred during Desert Storm with the noncombatant action in the Philippines, Operation Fiery Vigil, requiring simultaneous support from USTRANSCOM.

Knowledge Based Systems Engineering (KBSE)

The focus of the KBSE thrust is on the development, exploitation and demonstration of technology and tools to support design and implementation of robust, real-time, large-scale knowledge-based systems. This includes facilitating the use of advanced interface technology in complex C3I applications requiring natural modes of expression and a deeper level of interaction between the system and the user. The goal is to move from systems with a single type of information representation and reasoning strategy to designs which integrate multiple intelligent system schemes, integrated with conventional computing algorithms where appropriate, and on a much larger scale than currently possible.

Under a current effort, a testbed environment for design, rapid integration and evaluation of large scale knowledge-based systems is being developed. The Advanced AI Technology Testbed (AAITT) will support rapid integration of heterogeneous knowledge-based and conventional subsystems and will include capabilities for instrumentation and comparative analysis of alternative schemes. It will provide the Air Force a facility for developing and testing solutions to complex decision support systems involving the integration of knowledge-based and conventional software modules as part of the system design. Central to the AAITT concept is the KBSE's Knowledge-Based simulation research in-house effort. This R&D is concerned with the development and demonstration of advanced simulation techniques, providing a more flexible and dynamic environment needed for "what if" training and the exercising of integrated decision aid components.

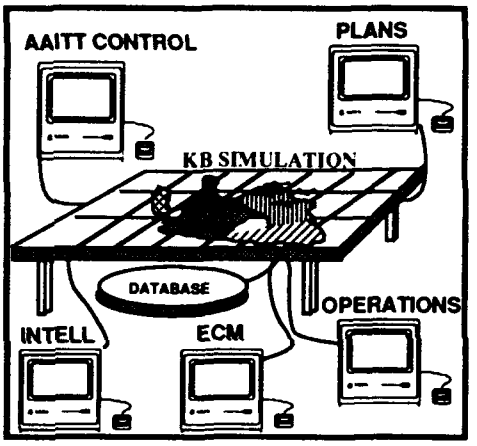


Figure 2. AAITT Testbed Concept

AAITT CONCEPT

Another effort is attempting to promote and facilitate the use of advanced interface technology and natural language processing in future complex and intelligent Air Force applications. Interfaces to AI systems must become more transparent to the user allowing natural modes of expression and a deeper level of interaction to take place between the system and user taking advantage of the intelligent capabilities of each. This activity addresses not only the issues which will enable optimal interface design, but also the practical aspects which allow designers to use advance interface technology in systems presently being developed. Natural language processing technology is being explored for use in explanation capabilities of knowledge based systems and natural language understanding of intelligence messages.

Under the KBSE thrust, several tools have been developed and demonstrated including a reasoning with uncertainty tool, the AAITT, and tools for reasoning about models and exploitation of parallelism. Techniques for acquisition and management of large scale knowledge bases have been embodied in tools developed under this program.

Three demonstration systems with incremental upgrades are planned for the AAITT. The first, delivered to the government in September of 1991, implemented a core C2I testbed, which included a mission planner, ORACLE database, and Tactical simulation, on top of a distributed processing substrate that allowed for flexible interchange of component subsystems. The second system, delivered in the September of 1992, includes measurement, instrumentation and monitoring capabilities and will be demonstrated solving a significant tactical C2I problem. The third and final system, scheduled for delivery in the fall of 1993 will demonstrate the testbed capabilities on a domain outside of C3I and will include modeling capabilities that allow the application developer to " test drive" a system before it is actually built.

Knowledge Based Software Assistance(KBSA)

In 1982, the Rome Laboratory (formerly the Rome Air Development Center) initiated a program to develop a knowledge-based system addressing the entire software system life cycle. The Knowledge-Based Software Assistant (KBSA), a retreat from pure automatic programming, is based upon the belief that by retaining the human in the process many of the unsolved problems encountered in automatic programming may be avoided. It proposes a new programming paradigm in which software activities are machine mediated and supported throughout the life cycle. The underlying concept of the KBSA, described in the original 1983 report entitled, "Report on A Knowledge-Based Software Assistant," is that the processes in addition to the products of software development will be formalized and automated. This enables a knowledge base to evolve that will capture the history of the life cycle processes and support automated reasoning about the software under development. The impact of this formalization of the processes is that software will be derived from requirements and specifications through a series of formal transformations. Enhancement and change will take place at the requirements and specification level as it will be possible to "replay" the process of implementation as recorded in the knowledge base. KBSA will provide a corporate memory of how objects are related, the reasoning that took place during design, the rationale behind decisions, the relationships among requirements, specifications, and code, and an explanation of the development process. This assistance and design capture will be accomplished through a collection of integrated life cycle facets, each tailored to its particular role, and an underlying common environment.

The goals of the KBSA program, as stated in the 1983 report, are to provide an environment where design will take place at a higher level of abstraction than is current practice. Knowledge-based assistance mediates all activities and provides process coordination and guidance to users, assisting them in translating informal application domain representations into formal executable specifications. The majority of software development activities are moved to the specification level as early validation is provided through prototyping, symbolic evaluation, and simulation. Implementations are derived from formal specifications through a series of automated meaning preserving transformations, insuring that the implementation correctly represents the specification. Post deployment support of the developed application system is also be concentrated at the requirements/specification level with subsequent implementations being efficiently generated through a largely automated "replay" process. This capability provides the additional benefit of reuse of designs as families of systems can spawn from the original application. Management policies are also formally stated enabling machine assisted enforcement and structuring of the software life cycle processes.

The techniques for achieving these goals are:

Formal representation and automatic recording of all the processes and objects associated with the software life cycle.

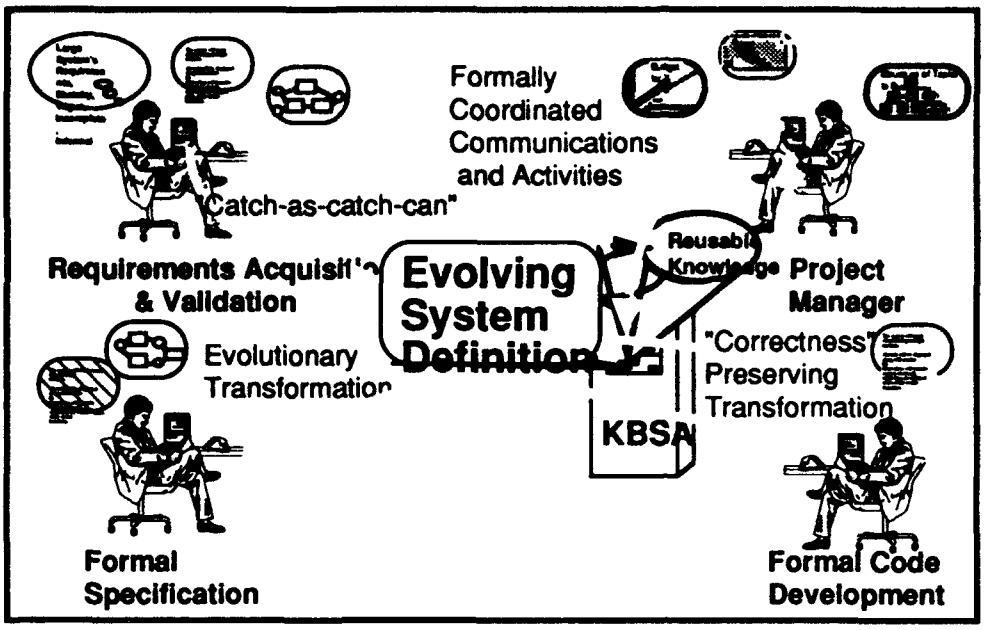


Figure 3. KBSA Concept Model

An Extensible knowledge-based representation and inference capability to represent and utilize knowledge in the software development and application domains.

A wide-spectrum specification language in which high-level constructs are freely mixed with implementation-level constructs.

Correctness preserving transformations that enable the iterative refinement of high-level constructs into implementation-level constructs as the KBSA carries out the design decisions of the developer.

The strategy proposed in 1983 to achieve the goals of the KBSA was to first formalize each stage of the present software life cycle model, with parallel developments of technology and knowledge bases for each particular stage. Supporting technology was also to be the subject of concurrent research and development efforts with periodic integration efforts or "builds" to assess progress and identify deficiencies. Although resource limitations have precluded the multiple parallel research thrusts of a magnitude originally proposed, initial products of the program have emerged with the successful completion of efforts which model and automate requirements definition, system specification, performance optimization and project management. Supporting technology has also been investigated and defined which will form the core of the KBSA and will be used in merging and managing the activities and processes of the various users. The following paragraphs will provide a brief description of the basic approach of each research effort and resulting products.

The first area to be addressed by the KBSA program was that of project management. In 1984 our program began defining a Project Management Assistant formalism and constructing a working prototype. The effort goals were to provide knowledge-based assistance in the management of project planning, monitoring, and communications. Planning assistance enables the structuring of the project into individual tasks and then scheduling and assigning these tasks. Once planned a project must be monitored. This is accomplished through cost and schedule constraints and the enforcement of specific management policies. PMA also provideses

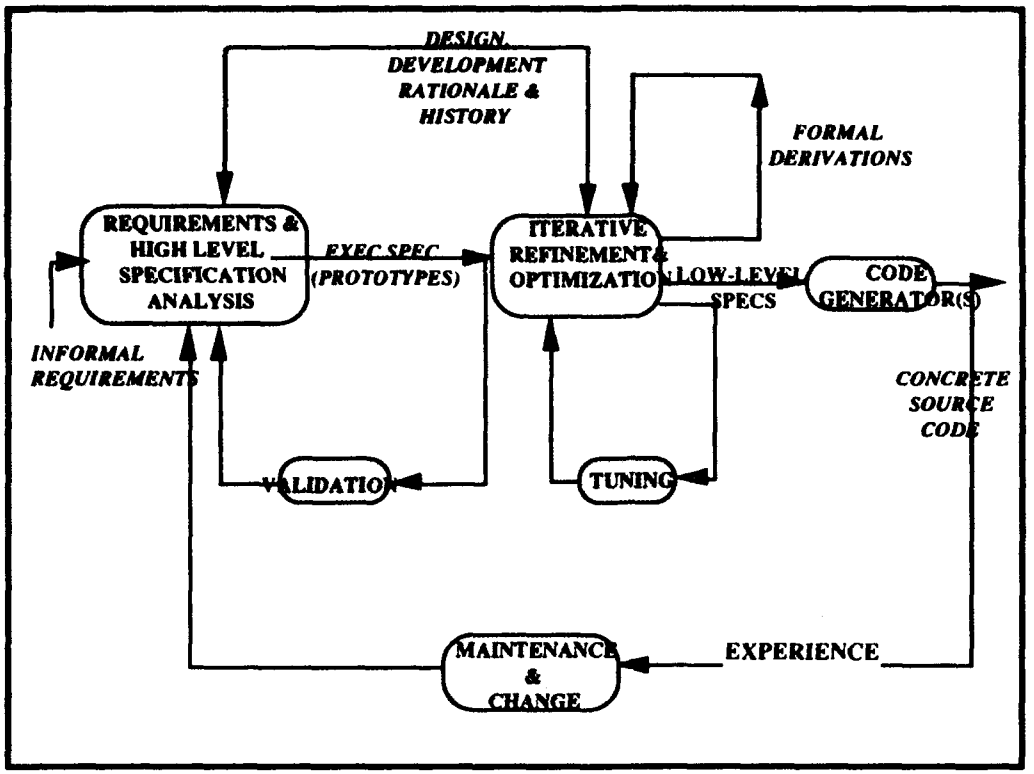


Figure 4.KBSA Process Model

user interaction in the form of direct queries/updates and various graphics representations such as Pert Charts and Gantt Charts. PMA is distinguished from conventional management tools because not only does PMA handle user defined tasks, but it also understands the products and implicit relationships among them (eg. components, tasks, requirements, specification, source code, test cases, test results, and milestones). Contributions of the PMA include the formalization of the above objects, the development of a powerful temporal representation for dependency relationships between software development objects, and a mechanisms for expressing and enforcing project policies. The initial PMA effort was completed in 1986. A subsequent PMA contract was initiated in November 1987. Its goals were to continue the evolution of PMA, expanding the formalized knowledge of project management to provide enhanced capabilities and to implement PMA as an integral part of a full-scale conventional software engineering environment called SLCSE (Software Life-Cycle Support Environment). This work recently completed in the summer of 1990.

In 1985 began work on the Knowledge Based Requirements Assistant (KBRA). Central to the KBRA was dealing with the informal nature of the requirements definition process including incompleteness and inconsistency. See Figure 3 above. In the KBRA environment requirements are entered in any desired order or level of detail using one of many differing views of the application problem. The KBRA is then responsible for doing the necessary bookkeeping to allow the user to manipulate the requirements while it maintains consistency among requirements. Capabilities included in the KBRA are support for multiple viewpoints (eg. data flow, control flow, state transition, and functional flow diagrams), management and editing tools to organize requirements, and the support for constraints and requirements that are not functional in nature through the use of spreadsheet and natural language notations. Other capabilities of the KBRA include the analysis of requirements to identify inconsistency and incompleteness, and the ability to generate explanations and descriptions of the evolving system. As previously indicated, the primary issue addressed by the KBRA was handling the informality of

incomplete user descriptions while building and maintaining a consistent internal representation. This was accomplished through the use of a representation providing truth maintenance support including default reasoning, dependency tracing, and local propagation of constraints. Through these mechanisms the KBRA was able to provide an application specific automatic classification which is used to identify missing requirements by comparing current input against existing requirements contained in the knowledge base.

Development of the KBSA Specification Assistant began in 1985. The goal of the Specification Assistant is to facilitate the development of formal executable specifications of software systems, a task that otherwise is as difficult as actually writing system code. It supports an evolutionary activity in which the system specification is incrementally elaborated as the user chooses among design alternatives. An executable formal specification language combined with symbolic evaluation, specification paraphrasing and static analysis allow early design validation by providing the user an evolving prototype of the system along with English descriptions and consistency checking throughout its design. Specification Assistant capabilities are built on the Lisp based AP5 language and the CLF development environment and utilize a variety of tools to support the user. A formal specification language called Gist is supported by a number of facilities which aid the development of specifications. One facility peculiar to the Specification Assistant is the support for specification evolution in the form of high-level editing commands, also known as evolution transformations. These commands perform stereotypical, meaningful changes to the specification. They differ from conventional "correctness-preserving" transformations in that they are specifically intended to change the meaning of specifications, but in specific ways. In addition to the top down evolution of specifications supported by the high-level editing commands, the Specification Assistant supports the building of a specification from smaller specifications (i.e.. the reuse of previously defined specifications) with a set of view extraction and merging tools.

In 1988 Rome Lab initiated an effort to combine the requirements acquisition process with that of formal system specification. This effort merges the developments of the KBRA and Specification Assistant, spanning all the activities needed to derive a complete and valid design from initial user requirements in one system called ARIES.

The development of an assistant to guide designers in performance decisions at many levels in the software development cycle was the goal of a 1985 contract with one of a small cadre of research institutes showing commitment and expertise in this area. This research produced a prototype system which takes as input a high level program written in a wide-spectrum language and following a combination of automatic, performance-based, and interactively-guided transformations, produces efficient code. The Performance Assistant supports the application of a variety of analysis and optimization techniques broken into the two general categories of control optimizations and data optimizations. The control optimizations include finite differencing, iterator inversion, loop fusion, and dead code elimination. Data optimization includes data structure selection, which implements a program's data objects using efficient structures, and copy optimization, which eliminates needless copying of large data objects. A subsequent effort to develop an independent tool that would assist in the development of high performance Ada software was initiated in 1991. This effort seeks to enhance the capabilities of the earlier effort by making them more robust and portable to more conventional software development environments, and by enabling the design and generation of optimal Ada code. The product of this effort is scheduled for delivery at the beginning of 1994.

An effort to define the requirements for a Framework sufficient to support the many varying facets of assistance provided by the KBSA commenced. The goal of this effort was to define a unifying framework which would support an object base with a tightly integrated logical inference system, configuration management, activity coordination, and user interface for the KBSA. This Framework sought to provide a common reference for other facet developers which when followed would allow the sharing of information. Also included in the effort was the goal of demonstrating the integration of multiple KBSA facets and the specification of common support capabilities. This effort resulted in: a Framework based upon a merging

of the Common Lisp Object System (CLOS) with the LogLisp programming environment; the KBSA User Interface Environment (KUIE); and, a preliminary Configuration and Change Management (CMM) model for the KBSA. LogLisp is a language developed at Syracuse University that extends a total Lisp environment with logic programming capabilities. KUIE is a highly object oriented system based on CLOS and the X11 Window System for constructing user interfaces.

The creation of an Assistant to support the transforming of formal specifications into low level coded was the goal of an effort started in 1988. The Development Assistant, sharing many capabilities with the Performance Assistant, is based the Kestrel Interactive Development System (KIDS) and is written in the Refine language. The system supports the construction of formal model of the application domain including the specification of the target system's desired behavior and the application of transformations to the specification to produce detailed code. The provided set of transformations encode design and optimization knowledge, allowing the user to mechanically make high-level design decisions which the system systematically applies. A facility is also provided which records derivations and provides the basis for future "replay", a fundamental concept of the KBSA. The Development Assistant was delivered to the Rome Laboratory in late summer 1991.

One of the original concepts which distinguished the KBSA was that of activity and communication coordination. This supporting technology was the subject of an effort undertaken by Software Options in 1988. The goal of this research was to define a formalism with a graphical syntax that could be used to specify and enforce the coordination of the many KBSA activities and communications allowing the programming of the KBSA processes. This effort resulted in the development of "Transaction Graphs," a formalism for specifying processes. Related to activity coordination is the problem of change and configuration management. In mid 1991, Software Options began the task of merging the formalisms for activity coordination and configuration management. Using Transaction Graphs and their existing "Artifacts" configuration management system they are developing a unified formalism for coordinating and managing the products and processes of the KBSA paradigm.

In 1988 the development started on a total life cycle demonstration of the concepts of the KBSA. This development provides a broad concept coverage but a shallow functionality demonstration capability for a narrow problem domain. The current KBSA Concept Demonstration system combines preliminary capabilities from the ARIES, Development, and Project Management assistants and includes example developments from an Air Traffic Control domain. It allows the demonstration of refinement of requirements and specifications, the complete capabilities of the Project Management Assistant including the automatic creation of tasks as the design progresses, and the automatic generation of Lisp code from the specifications. Many additional capabilities are available for examining and manipulating both informal and formal representations of design including hypertext, multiple graphical representations, English like explanations, and the simulation of application system execution. The final product, delivered in October of 1992, also addresses the formal verification of specifications.

Current program activities include the development of an initial operational capability, the KBSA Advanced Development Model (ADM), and a broad spectrum of research directed toward technology and capability deficiencies. The ADM will be the first attempt at integrating the KBSA technologies to form a working environment. The work includes design and development of a robust environment of acceptable performance combined followed by evaluation through development of a moderate sized "real" application. This work will begin in December of 1992 with completion four years later. Award of a range of efforts arising from the current Program Research and Development Announcement (PRDA) is anticipated in the spring of 1993. These efforts will continue the evolution and refinement of KBSA technology and it is hoped that close coordination of these efforts with that of developing an operational capability will accelerate both technical accomplishment and acceptance.

Although the KBSA is much closer to fruition than true "automatic programming" and much optimism exists as evidenced above, it is an ambitious project and sought after results should not be expected soon. Future efforts of the KBSA program include the development of an operational KBSA system starting in 1992

and the continued evolution of existing components and supporting technology. The desire for more immediate benefits has been addressed by producing "spin-off" tools for conventional environments, hosting annual workshops, and forming the KBSA Technology Transfer Consortium providing industry immediate access to the technology and tools of the program. The goal of these activities is to attain an initial operational capability of the KBSA by late 1996 or early 1997.

Summary

Rome Lab's program is attempting to enhance current AI technology for use in large, real-time mission critical systems and developing the software tools that improve and enable the development, fielding and maintenance of these AI-based systems. This program is addressing critical technology shortfalls with respect to Air Force mission needs and advancing that technology in the three thrust areas of knowledge based planning, knowledge based systems engineering and knowledge based software assistance. Evaluation and demonstrations of the technology and tools has and will continue to be performed in the context of C3I mission functions.

References

Green, C., D. Luckham, R. Balzer, T. Cheatham, and C. Rich. "Report on a Knowledge-Based Software Assistant", Rome Air Development Center report RADC-TR-83-195, Aug 1983.

Jullig, R., W. Polak, P. Ladkin, and L. Gilham, "KBSA Project Management Assistant", Rome Air Development Center report RADC-TR-87-78, Jul. 1987.

Harris, D. and A. Czuchry, "Knowledge Based Requirements Assistant", Rome Air Development Center report RADC-TR-88-205, Oct. 1988.

Johnson, W., D. Cohen, M. Feather, D. Kogan, J. Meyers, K. Yue and R. Balzer,"KBSA Specification Assistant", Rome Air Development Center report RADC-TR-88-313, Feb. 1989.

Goldberg, A., L. Blaine, T. Pressburger, X. Qian, T. Roberts and S. Westfold,"KBSA Performance Estimation Assistant", Rome Air Development Center report RADC-TR-89-98, Aug. 1989.

Huseth, S. A. Larson, J. Carciofini and J. Glasser, "KBSA Framework", Rome Air Development Center report RADC-TR-88-204, Oct. 1988.

Huseth, S. A. Larson, J. Carciofini and J. Glasser, "KBSA Framework", Rome Air Development Center report RADC-TR-88-204, Oct. 1988.

Larson, A. J. Kimball, J. Clark, R. Schrag, "KBSA Framework", Rome Air Development Center report RADC-TR-90-349, Dec. 1990.

Daum, M. R. Jullig, "Knowledge-Based Project Management Assistant for Ada Systems", Rome Air Development Center report RADC-TR-90-418, Dec. 1990.

Karr, M., "Transaction Graphs: A Sketch Formalism for Activity Coordination", Rome Air Development Center report RADC-TR-90-347, Dec. 1990.

Cross, S. "A Proposed Initiative in Crisis Action Planning" unpublished white paper, DARPA/ISTO Arlington VA. May 18 1990.

Discussion

Question D. NAIRN

How focussed a solution is DART? Has it any general application? What validation is needed before operational use?

Reply

In general, DART is simply a graphical interface to an ORACLE database. In particular, it focuses on and displays time windows : early, latest start/arrival dates and transportation durations. The simulation aspects of DART is a general feasibility analysis tool for determining the percentage of on-time arrivals.

Validation, due to the extremely short development time, was stress testing and user participation during development.

Question D. NAIRN

Have you studied what percentage of maintenance tasks involve changes to the specifications and what percentage are retricted to implementation, eg hardware variants, bug fixes, etc?

Reply

In short, no. But my general feeling is that variants and configurations should be documented as requirements and hence become part of the specification as appropriate.

No bugs means no bug fixes. "Bugs" that are a fault of the code-generation process would indicate a fault in the KBSA (or in rare cases, in the hand-coding : this should be avoided!). Other bugs may be results of incomplete or inconsistent requirements. These types of bugs should be detected prior to coding, although incompleteness of requirements is difficult to detect.

Potential Software Failures Methodology Analysis

M. Nogarino
D. Coppola
L.Contrastano
ALENIA - DAAS
Viale Europa
20014 Nerviano (MI) ITALIA

SUMMARY

ALENIA - Nerviano Plant is mainly involved in the development of complex avionic systems, of which software is essential component, often presenting safety critical features. Inadequacy of the traditional techniques and methods impose to support them with additional refinement tools to build the product safety during developing phases.

The paper describes a methodological approach for the systematic identification and classification of the effects caused by potential software failures. The potential software failures are identified evaluating the effects that would be produced by incorrect outputs on the other software parts and on the external environment. Furthermore, the proposed approach allows to evaluate the necessity to introduce fault-tolerance, recovery and backup procedures, and to define the adequate quantity and typology of testing and quality activities.

1. INTRODUCTION

The paper describes a qualitative methodology, utilized in Alenia, to face systematically the identification and classification of the effects caused by potential software failures and details its operating steps, which mainly consists of: functionality analysis and architectural design, potential failure mode identification, effects evaluation, criticality categories assignation, and corrective actions identification.

The utilized methodological approach consists essentially of the analysis of each software requirement and of each architectural component, with the purpose to evaluate the effects that would be produced by incorrect outputs on the other software parts and on the external environment.

As a result of the inherent complexity of the software development process, developers have plenty of opportunities to make errors. The total number of defects injected into software not intentionally by analysts, designers and programmers from requirements determination to delivery is very large. Most of these errors are removed before delivery through self-checking by analysts and programmers, design reviews, walkthroughs, inspections, testing.

The effort to remove and prevent defects before delivery is more intense in the case of application types where the error has serious consequences, even life and death, than it is in applications where the consequences are less drastic.
Reducing errors is, of course, an extensive effort, embracing every aspect of software development.

The current methods, utilized for the software production in applications with criticality characteristics, consist of activities of development and control during the whole life cycle, and affect considerably cost and time.
They interest uniformly all the software functionalities, without distinguishing between the elements.

The actual aim is to minimize the most critical functionalities defects only, and to operate during the requirements analysis and design.
The critical meaning is defined regarding the actual context, then there will be defined critical the functionalities with effect on mission objectives, on human life, on financial objectives, or on environment, et cetera.

So, the methodology is of general applicability, even if below its description refers to a context in which the software functions criticality is identified basing on the safety.

Linking with the avionic equipments failures nature, there will be the software risk class 1, 2 and 3.

Consequently, the software failures involving in the software risk classes have an analog classification: failure class 1, 2 and 3.

2. PROCEDURE DESCRIPTION

The methodology requires as background the quality standard concepts, among which the Alenia Quality Manual appears and consists of the principal military and civil rules guide lines.

The methodology aim is to support the development of software for a safety-related system.
Furthermore, it starts from the initial hypothesis which takes for known the output characteristics, that go from the software environment to the External Interface. In detail, it must be known before the criticality risk class of all the outputs towards the External Interface. The risk class may be 1, 2 or 3.
Basing on the output risk classes, the functionalities and CSUs risk classes are assigned, during the software requirements

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

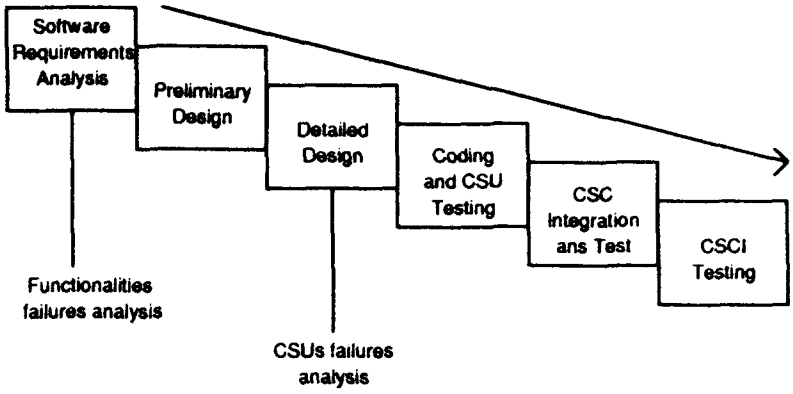


Figure 1 - Software development process

analysis and the detailed design phases respectively.

The methodology becomes part of the software life cycle (Figure 1). It is applied during the analysis and detailed design phases profitably, while it furnishes useful information for the choice of the strategy to adopt during the coding and test phases.

2.1 Software Requirements Analysis Phase
(Functionalities failures analysis)

In this phase the functionalities risk classes are identified and then an eventual fault-tolerance mechanism is introduced.

At the beginning, the software requirements are identified and analysed. The Structured Analysis allows to do that, because it puts particular attention to the input and output data flows, utilized in the next steps.

Another analysis may be utilized, on condition that it points out the data flows; or, an already realized analysis, that follows the previous requirements, may be utilized however.

The analysis continues until the basic functionalities are identified.
The analysis is satisfactory when, for each identified functionality, a description of what is required to be executed is furnished. So, at the analysis end, it is possible to obtain a complete vision of the data flows and how they have to be treated; furthermore, it is possible to furnish a first sign about the control flows, in fact sometimes the information treatment sequence is pointed out.

Now, the steps to be followed are described and the Structured Analysis is applied.

When the wished detail is reached, the risk class 1 and 2 outputs towards External Interface are detected.
The risk class 3 outputs aren't considered.

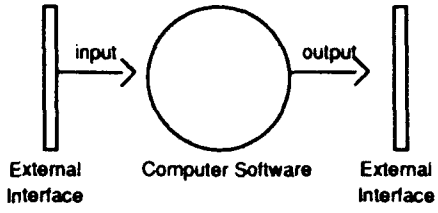


Figure 2 - Level 0

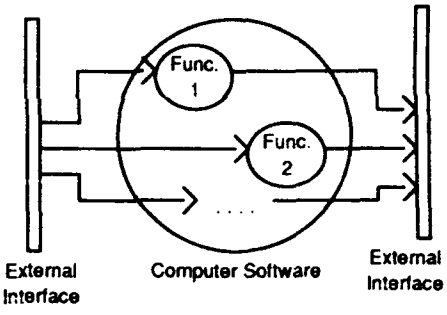


Figure 3 - Level 1

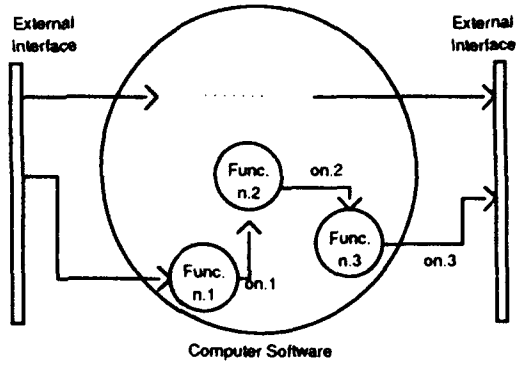


Figure 4 - Level 2

For each identified output, the set of functionalities that lead to the output creation is pointed out; in the example one set consists of the Functionalities n.1 - n.2 - n.3, because on.3 is of risk class 2.
The functionalities sets may be reduced furthermore, if only the functionalities that lead to the risk class 1 outputs are considered. This choice must be done basing on the specific requirements of each project.

For each previous identified functionalities set and starting from the first functionality, the following steps are fulfilled:

1. Determine if an unexpected output from the considered functionality or the output missing when expected can affect the last output.

   This consideration is realized starting from the initial hypothesis that the other functionalities run properly and basing on the knowledge of what is the behaviour required for the functionalities, that are executed after the last one considered.

2. If the answer is positive, the considered functionality inherits the last output risk class (1 or 2).

3. If the considered functionality is of risk class 1 or 2, then it is possible to introduce new fault-tolerance requirements, to manage the potential failures.

   After the fault-tolerance mechanism has been introduced, the functionality risk class can decrease. In this case, it is marked with an asterisk.
   All the functionalities, that are previously analysed and involved in the fault-tolerance mechanism, are evaluated again.

4. If the considered set contains other functionalities, then consider the next functionality and restart from the point 2.1.1; otherwise, execute the steps from 2.1.1 to 2.1.4 on the next set, until all the previously identified sets are considered.

After the steps from 1 to 4 have been executed for each functionalities set, it may occur the need to check the analysis, or part of it, so that the basic functionalities are reorganized basing on their risk class, isolating, where it is possible, the risk class 1 or 2 functionalities sets.
At this time, if it is necessary, ,the steps from 1 to 4 are reapplied on the just identified functionalities sets.

2.2 Detailed Design Phase
(CSUs failures analysis)

During the detailed design phase the risk classes of the CSUs are detected.

It is suggested to apply the methodology during this phase, also if it has just been applied during the requirements analysis. In fact, it allows to distinguish the CSUs with greater risk class (1 or 2) furthermore, improving the previous analysis result.

From the requirements analysis it passes to the preliminary design, conforming with the obtained results. In this phase the methodology isn't applied, because it doesn't add relevant information respect the previous phase.

From the preliminary design, it passes to a detailed design, that defines the CSUs leaves. The Structured Design is indicated for this scope, but another design type may be utilized too, in which the data and control flows are pointed out.

As in the requirements analysis phase, also in this case a previously executed design may be utilized, on condition that it fulfills the flows requirements.
The design is satisfactory when, for each CSU, it provides a description of the data management and control.

Now, the steps to be followed are described, and the Structured Design is applied.

When the wished detail is achieved, the risk class 1 or 2 outputs towards the External Interface are pointed out.
The risk class 3 outputs aren't considered.
The output risk class must be defined conforming with the analysis definitions.

For each identified output, the sets of CSUs that lead to the output creation are pointed out; in the example, one set consists of the CSUs 1.1. - 1.1.2 - 1.2 - 1.3, because the

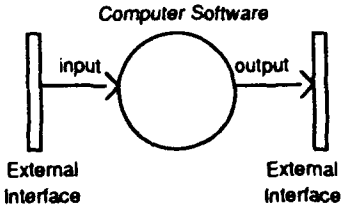


Figure 5 - Level 0

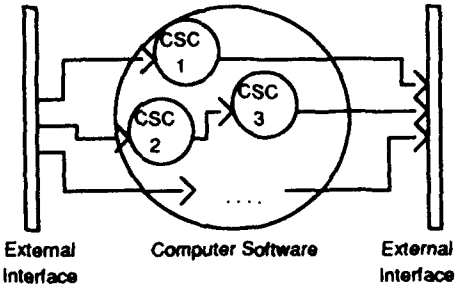


Figure 6 - Level 1

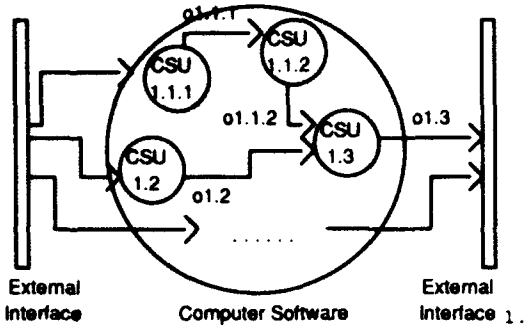


Figure 7 - Level 2

output o1.3 is of risk class 2.
Specific project demands can require to reduce the number of the CSUs sets to be considered, defining more restricted constraints.

For each previously identified CSUs set, starting from the first CSU, the following steps are fulfilled:

1. Determine if an unexpected value of the output, or the output missing when expected, or the unexpected exit of the output, can affect the final output considerably.

   Furthermore, while at requirements analysis level all cannot be defined, at design level the information treatment description is complete.

2. If the answer is positive, the considered CSU inherits the last output risk class (1 or 2).

   If the considered CSU risk class is 1 or 2, a new row in APOSF (Analysis of the POssible Software Failures) Table is initialized.

3. If the considered CSU risk class is 1 or 2, the possible failures are determined. More precisely, the generic failure situations are:

   a) loss of the function identified by the CSU; for example, the function has not been activated for a scheduling error;
   b) running of the function identified by the CSU when it is not requested; for example, there is an erroneous control flow, or there are temporization or synchronization errors;
   c) the function identified by the CSU doesn't work correctly; for example, an implementation fault.

   All the hypothetical failures, the effects of each failure on the other CSUs and on the external Interface, and the affected CSUs have to be written in APOSF Table.

4. After the possible failures detection, when the considered CSU risk class is 1 or 2, it is possible to introduce fault-tolerance mechanisms.

   The detected solutions and the interested CSUs are reported in APOSF Table.

   Choosing between the thought solutions, the more suitable one for the current situation is identified, and, basing on it, the considered CSU risk class is evaluated again.
   Furthermore, if the adopted solution modifies previously analysed CSUs, it needs to evaluate again their risk class; if their risk class decreases, mark it with an asterisk. After, verify their failures effects and eventually bring up-to-date APOSF Table.

5. If the set contains another CSU to analyze, consider it and restart from point 1; otherwise, apply the steps from 1 to 5 to another CSUs set, until the identified sets finish.

After the steps from 1 to 5 are executed for all or part of the CSUs sets, it may occur the need to reorganize the design, or part of it, so that also the CSUs are reorganized basing on their risk class and, where it is possible, the risk class 1 and 2 CSUs sets are isolated.

Figure 8 - APOSF Table

Order Numb.	CSU (1	CSU descr. (2	Functio-nalities (3	Risk class (4	Failures (5	Failures effects (6	Fail.-rel. CSU (7	Solutions (8	Rel. CSU (9
Note:									(10

At this point, if it is necessary, the steps from 1 to 5 are reapplied on the just identified CSUs sets.

2.3 Other Phases

In the Coding and CSU Testing Phase, CSC Integration and Testing Phase, and CSCI Testing Phase, the information present in APOSF Table are a useful means to plan the activities.

Indeed, at the coding phase it is possible to select the language to be used for each software part or the algorhitms to be adopted, so that the response time requirements or other critical constraints, identified during the previous phases, are fulfilled.
Furthermore, a scrutiny activity may be executed on the more critical points for the safety.

During the CSU testing phase, the CSUs sets to be considered are identified, basing on the risk class (1, 2 and 1*, 2*).
In fact, the information furnished during the design phase allow to identify the CSUs to be controlled much more, and to point out the more meaningful input data, so the testing time is reduced and the safety is safeguarded.

Also in this CSC Integration and Testing Phase, the information furnished during the design phase allow to identify the most critical software parts to be considered; so, the CSUs relations, the Interfaces between the CSCs and the External Environment, and the Interfaces between the same CSCs are tested.

During the CSCI Testing Phase the information obtained from the requirements analysis phase may be utilized to point out the most critical functionalities and define the opportune input sequence and the expected outputs.

So, the effort is concentrated on few aspects and the safety is safeguarded equally.

3. EXAMPLE

The procedure has been applied on an avionic equipment, which transmits data between the operator and the internal system, and consists of hardware and software parts. More precisely, it has been considered the equipment software part which manages the information coming from the Front Panel and the relative information stored in the internal memory.

3.1 Functionalities failures analysis

The software requirements analysis has been done using CORE (COntrolled Requirements Expression), which identifies the input and output data flows, and then may be considered for the actual context.

It has been pointed out a final output of risk class 2, that is a Flight Safety Involved Signal.

The sequence of basic functionalities that create this output has been identified. Shortly, one functionality tests if there is a new input from the Front Panel Interface; if the answer is positive, the next functionality converts the data from the Front Panel format into the final output. Afterwards, the new converted data are loaded by another functionality into another memory part.

By applying the methodology, four new failure situations have been pointed out. Indeed, for example, it is not usually considered the possibility in which the called functionality doesn't produce any output.

More precisely, it has been verified that the output generated by the first functionality, that tests the presence of new data from the Front Panel Interface, cannot affect the final output, and it always exits when the functionality has been called.

With regard to the last two functionalities, it exists the non anticipated possibility in which the output isn't produced or is produced with an unexpected value. The methodology application has furnished the possibility to prevent this failure situation, by introducing the opportune fault-tolerance mechanisms.

3.2 CSUs failures analysis

The software design has been made using the SD (Structured Design) tecnique.
It has been pointed out a final output of risk class 2, which is the Flight Safety Involved Signal identified during the previous analysis phase.

Shortly, there is a CSU to test the presence of new data from the Front Panel Interface; there is a CSU for the data copy from a memory part to another.

The procedure application created APOSF Table as partially displayed in Figure 9.

4. ADVANTAGES

As previously described, the methodology considers the failure situations caused by incorrect elaboration, synchronization errors, scheduling errors. It permits to obtain a complete analysis of the possible failures.

The methodology application also at detailed design level allows to evaluate the possible failure situations dependent from the project context (e.g., Operative System, Basic Software, memory dimension, CPU clock rate, etc.).

So, the principal characteristic of the methodology is that, besides the most critical functionalities and CSUs, the

Figure 9 - APOSF Table

Order Numb.	CSU	CSU description	Functio_ nalities	Risk class	Failures	Failures effects	Fail.-rel. CSU	Solutions	Sol.-rel. CSU
1	CHECK_ REQUEST _DKC	It verifies if conversion requests are present on the FP IF.	REQ_ D01221	2->3*	.....				
2	CONV_ FP1KEY	It converts data from FP1 format in Stanag 3838 format.	REQ_ D01222	2	An unexpected output value is produced.	The final output takes on an unex_ pected value.	LOAD_ SUBADD03	A filter is introduced in the next CSU.	LOAD_ SUBADD03
					The output isn't produced when it is expected.	The new final output is lost.	LOAD_ SUBADD03	Sequencial treatment of the data in the CSU CONV_ FP1KEY	CONV_ FP1KEY
3	LOAD_ SUBADD03	It loads data to subaddress 03 of Stanag 3838 IF.	REQ_ D01223	2->3*	.....				

functionalities and CSUs which contain error filters or other fault-tolerance mechanisms are considered with greater attention. Only these parts are considered, and then the software to be studied carefully decreases in dimension.

By identifying the critical parts, the methodology allows to support the project choices, by developing procedures that manage the errors and recover the failure situations; and, it allows to minimize the testing activities, by planning the functional and structural tests and the relative covering requirements; and, it allows to point out those software parts on which the walkthroughs and the inspections have to be made.
So, the methodology allows to obtain a quality product with reduced time and resources cost.

The introduction of opportune fault-tolerance mechanisms increase the reliability of the software part, and then, of the whole system. The fault-tolerance mechanisms have to be applied in a discriminated manner, looking at the maximum effect with the minimum alteration and additional code.

By identifying the most critical functionalities and CSUs, the methodology reduces considerably the number of functionalities and CSUs to be treated with particular attention, and then represents a useful means to integrate the traditional methods.

5. FUTURE DEVELOPMENTS

Sometimes, in particularly big projects, although the number of functionalities to be considered is considerably reduced, the introduced procedure requires a lot of time. For this reason and, in general, for a good application of the methodology, it needs to automate the methodology itself.

In this context, a weighted metric may be created, basing on the functionalities and CSUs risk classes. This metric may be applied at the end of the analysis and detailed design phases.
This is the next step to be achieved.

Furthermore, the methodology furnishes good rules to identify the most critical parts of the software. To obtain a quality product, it is necessary to adopt opportune fault-tolerance methodologies.
Afterwards, it follows the necessity to create an expert system, which evaluates the contingent demand and decides the fault-tolerance type to be adopted in the actual context.

This expert system may operate together with the previously hypothetical weighted metric.

6. REFERENCES

1. Musa,J.D and Iannino,A. and Okumoto,K., "Software Reliability", USA, McGraw Hill, 1987

2. Wichmann,B.A., "Software in Safety-related Systems, UK, John Wiley and Sons Ltd., BCS Wiley, 1992

3. Siewiorek,D.P. and Swarz,R.S., "Reliable Computer System - Design and Evaluation", USA, National Physical Laboratory, Digital Press, 2nd. Ed., 1992

4. "Eurofighter - Software Development Standards", PL-J-019-E-1006, Issue 2 Draft A, January 19921

5. "Defence System Software Development", DOD-STD-2167A, Military Standard, 29 February 1988

6. "Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment", 00-56 / Issue 1, Ministry of Defence, Interim - Defence Standard, 5 April 1991

7. "Software Reliability", MIL-HDBK-338, 15 October 1984

8. "Software Aspects of System Safety Assessment Procedure", PL-J-000-E-1020, Issue 1, April 1990

9. "The Development of Safety Critical Software for Airborne Systems", 00-31 / Issue 1, Ministry of Defence, Interim - Defence Standard, 3 July 1987

10. "System Safety Engineering Design Guide for Army Material", MIL-HDBK-764 (MI), 12 January 1990

11. "Software Reliability Requirements Analysis and Specification for ESA Space System and Associated Equipment", ESA PSS-01-230 / Issue 1 Draft 4, June 1989

Discussion

Question C. BENJAMIN

How is the methodology applied? Do it rely on inspection of the structural analysis and structural design charts?

Reply

The methodology has been applied on the graphs obtained with the structural analysis application, during the software requirements analysis phase, and on the graphs obtained with the HOOD application, during the design phase. We rae developping a tool to manage greater projects and to operate with the rsults obtained from other method application (eg CORE).

Question W. MALA

What definition has been used for the various risk classes, eg class 1, class 2, etc?

Reply

The risk class definition has been derived from RTCA DO 178A using the risk class terms "catastrophic", etc.
More precisely, the risk classes utilized in the paper are :

class 1 : an error affecting the human life and the environment,
class 2 : an error that can produce situation in which the human life is not safeguarded,
class 3 : an error that does not affect the human life safety.

ROME LABORATORY SOFTWARE ENGINEERING TECHNOLOGY PROGRAM

Elizabeth S. Kean
Rome Laboratory/C3CB
525 Brooks Rd
Griffiss AFB NY 13441-4505
United States

SUMMARY

This paper highlights the technology recently developed or under development in the US Air Force's Rome Laboratory Software Engineering Technology Branch. This program is generic in nature, focused around development and support of large, mission critical and embedded software systems, and thus is very relevant to the development and support of avionics systems. Further, when a technology is programming language sensitive or a demonstration vehicle requires selection of a specific language, the programm language selected is always Ada. Finally, this program has four major thrusts.

One thrust emphasizes system definition technology and is concerned with development and validation of requirements and specifications. A second thrust explores and builds technology for integrated software and system engineering environments, with emphasis on tools, process support and enforcement, and support to development and acquisition managers. New explorations in this area include certification methodologies and tools for reusable software components, and software fault-tolerance (robustness). A third thrust deals with the specification, prediction, and assessment of software quality. Rome Lab has a framework for dealing with all aspects of software quality that has proven itself in Japan. The newest thrust is on softare engineering for high performance computing. This unique program is evaluating and developing generic software methods and tools for using high performance, massively parallel computers in embedded and other mission critical applications.

INTRODUCTION

Rome Laboratory, formerly Rome Air Development Center (RADC), is one of the Air Force's four super-labs. It is headquartered at, Griffiss Air Force Base, which is adjacent to the city of Rome, New York. For over forty years, Rome Lab and its predecessor RADC, have been a major force in computer science and technology in areas such as processor and memory technology, compilers and programming languages, data bases, operating systems, artificial intelligence (AI) and decision aids, computer security, and software engineering technology. Rome Laboratory is the only Air Force Lab chartered to do "generic" computer technology. Research and development products have found their way into programs like the F-16, LANTIRN, PAVE-PAWS, WWMCCS, MX, PERSHING, and many more too numerous to mention. The basic premise is that automated software tools that support solid software engineering principles is the way to approach the productivity and quality problems from a technological perspective. Both DoD and Air Force studies of the "software crisis" cite concerns over burgeoning demand for software, lack of stability in requirements, shortages of skilled personnel, and high costs for software error correction.

The program is focused on all phases of the system and software life cycle from requirements analysis through code, test, integration, and post deployment support to fielded systems. We are faced with an ever diminishing defense budget and many of the systems in existence today will be around for some time in the future. Improvements to these systems to maintain a strong defense will be needed and the software engineering technology applied during their initial development must be augmented during the post deployment phase in order to assure continued success and mission compliance. The software engineering program consists of the following technology areas: System/Software Support Environments, System Definition Technology, Software and System Quality, and Software for High performance Computers.The allocation of the total software engineering program to these areas enables a focus on high payoff technology at key points in the life cycle. The extensive use of Air Force Materiel Command (AFMC) operational test and evaluation (Beta) test sites coupled with a Technology Transition Plan with the HQ Air Force Software Technology Support Center

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

(STSC) provide significant opportunities to assure program responsiveness to user needs and technology transition.

1. SYSTEM/SOFTWARE SUPPORT ENVIRONMENTS

The objectives in this area are to develop life cycle support capabilities for software intensive systems, to certify the reusability of software components, develop advanced test techniques for fault tolerant software, and to serve as the transition vehicle for Rome Laboratory software engineering products. Current work in the area has focused on an integrated software engineering life cycle framework called the Software Life Cycle Support Environment (SLCSE) (see Figure 1).

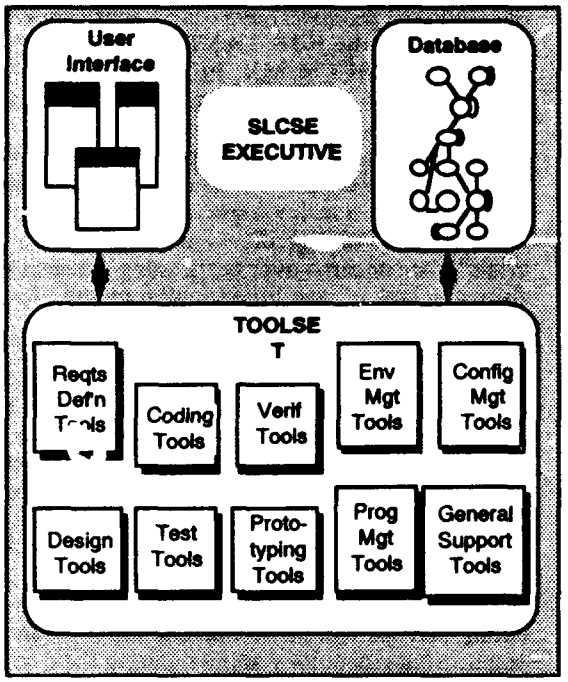


Figure 1
Software Life Cycle Support Environment (SLCSE)

The SLCSE allows a system developer or maintainer to capture the productivity and quality enhancements provided by today's Computer Aided Software Engineering (CASE) tools in a single environment, with a single common user interface and data base. SLCSE can be arbitrarily packed with tools to the degree the user wants. If used properly from the beginning of a development, SLCSE will automatically provide all the documentation required by DOD-STD-2167A. System engineering capabilities will be developed to augment the SLCSE and provide for hardware, firmware, and software development. Tools for functional decomposition of system requirements into their respective allocations will be conducted such that all system components will be accounted for along with the automated production of user documentation. The Environment supports a multiplicity of high order languages including Ada, FORTRAN, COBOL, and JOVIAL. A SLCSE Project Management System (SPMS) enables program managers to track software life cycle progress during development and to match effort expended against the work breakdown structure established, report on milestone activity, and to conduct critical path analyses. An Ada Test and Verification System (ATVS) is a component of the SLCSE tool set and is available to be applied during the development of and support to Ada software systems. SLCSE has undergone beta-testing in the aerospace community and was called "the" state-of-the-art environment by the late Dr Howard Yudkin, president and CEO of the Software Productivity Consortium.

Enhancements to the SLCSE resulting in an improved framework are ongoing which will significantly improve the common user interface and allow the database to communicate with several commercially available database management systems. System engineering concepts are being examined to determine the best way to incorporate system engineering tools and methods into the environment. The enhancements also provide a means to control and manage the process of software development and production and will be tailorable to unique organizational or mission needs. A distributed architecture will be employed to enable remote use of common terminals for text processing and message handling as well as sophisticated work stations for more complex and difficult software tasks.

The enhanced product called ProSLCSE is being funded by Rome Laboratory, the Electronic Systems Division, and the Strategic Defense Initiative. ProSLCSE provides a computer-based framework which may be instantiated to create an environment tailored to accommodate the specific needs of a software development or support project. A ProSLCSE environment supports a total lifecycle concept (i.e., concept exploration, demonstration and validation, and engineering and manufacturing development). A key feature of the ProSLCSE is the repository that contains all the accumulated information that can be transitioned to the Post-Deployment

Support lifecycle phase. The ProSLCSE Environment will be fully productized and supported with training, documentation, on-site and on-call assistance, and site specific installation and start-up. (Rome Laboratory Point-of-Contact: Mr. James R. Milligan, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-2054, DSN 587-2054)

Planned efforts include the development of a certification methodology and tools to enable software developers to determine a "level of confidence" in candidate software components identified as having reuse potential whether these components exist in a library or have been applied in like systems. Levels of certification will be based on user needs analysis and the desired/required confidence level sought. (Rome Laboratory Point-of-Contact: Ms. Deborah A Cerino, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-2054, DSN 587-2054)

Another effort will develop advanced test techniques for fault tolerant software systems and will address issues in software requirements analysis and design for fault tolerance which can be integrated with conventional fault detection and fault isolation techniques which have traditionally dealt with hardware base approaches. (Rome Laboratory Point-of-Contact: Mr. Roger J. Dziegiel, Jr., RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-2054, DSN 587-2054)

2. SYSTEM DEFINITION TECHNOLOGY

Recognizing that requirements errors are the most frequent (over 50% of all errors) and more expensive to correct the further they percolate through the system (up to 50 times more expensive to correct in system integration than in requirements analysis), the Rome Laboratory has focused on technology to catch those errors during the requirements phase itself. The process begins with the elicitation of user requirements whereby the user states operational requirements in terminology that the user is familiar with. The next step is to translate those requirements into a specification of what is required for the solution or system to be fielded and it is up to the acquisition agent to build a system and software that is responsive to the users needs. There are many opportunities for requirements to be misunderstood or incorrectly specified since the user is typically not directly involved in the process after the initial phases of the life cycle. The Rome Laboratory program in this area is concentrated on producing a Requirements Engineering Environment to enable end-item users to become involved in the requirements process, to provide techniques for automated code production, develop reusable specifications, and to integrate requirements engineering technology more fully with the life cycle process. The Rome Laboratory Requirements Engineering Environment (see Figure 2) attempts to overcome requirements oriented problems by keeping the user involved in the process and by providing the user with an early, and first hand, view of what the final product should look and feel like.

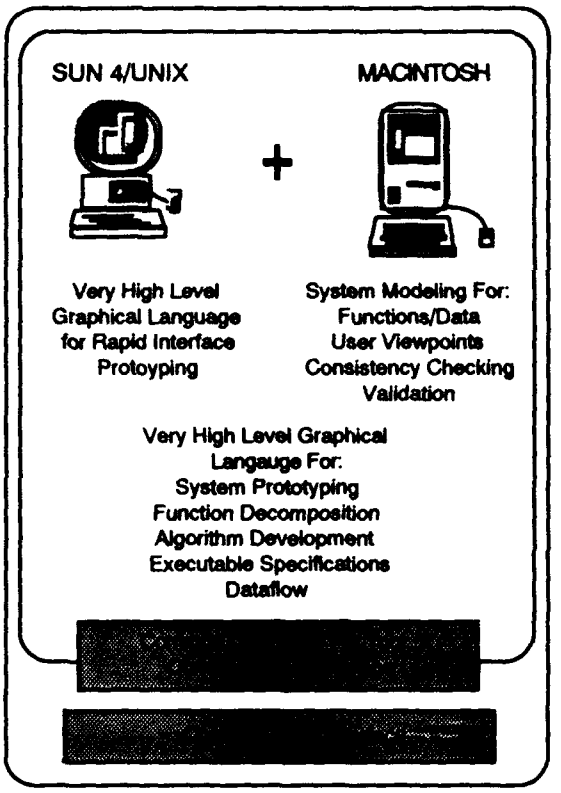


Figure 2
Requirements Engineering Environment (REE)

During the past ten years, Rome Laboratory developed and demonstrated tools to rapidly prototype the requirements for the displays, functionality, algorithms, and performance of a C3I system and to insure that all the individuals involved in developing specifications for the system have a consistent set of views on the system. This technology is being integrated into a single requirements engineering workstation environment. The primary integration vehicle will be the object management system. It will

store all information which is not in the exclusive domains of the individual tools, thereby allowing sharing of information which is needed by one or more of the other tools. The environment supports the research and development of methods and tools and the application of their application to realistic C3I system and software requirements problems and the evaluation of these methods and tools in terms of the productivity of the processes involved and quality of the products they produce.

Reusable C3I specifications are being addressed to examine their potential for post deployment block upgrades and application to other systems. Instead of code reusability (which may not meet performance requirements) the specifications for similar systems may be decomposed and assessed for reusability of specifications which have been verified and validated under operational conditions. (Rome Laboratory Point-of-Contact: Mr. William E. Rzepka, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-2762, DSN 587-2762)

In the future, the emphasis in requirements engineering technologies at Rome Laboratory will shift to considering more formal approaches which take advantage of matured AI technologies such as the Knowledge Based Software Assistant (KBSA). Since the early 1980's Rome Laboratory has been developing the KBSA as an alternative software development paradigm in which a formal executable system specification evolves through the elicitation and transformation of informal requirements expressed in representations familiar to the application scientist or engineer.

Planned work involves the development of a Advanced Requirements Engineering Workstation and the integration of the current Requirements Engineering Environment with the Process Oriented Software Life Cycle Support Environment (ProSLCSE). In the first effort, corporate knowledge and skills applied to previous systems may be brought to bear on new problems and system developments. Domain and community knowledge can be input to the requirements process to further assist the user in defining and specifying system and software requirements. The Advanced Requirements Engineering Workstation will make use of both conventional software engineering technologies and artificial intelligence approaches to develop an environment for modeling requirements which supports multiple external views of the requirements while maintaining a single consistent internal representation system which allows reasoning about the requirements. Work is currently underway to develop the architectural design for this environment. (Rome Laboratory Point-of-Contact: Mr. James L. Sidoran, RL/C3CB, 525 Brooks Rd., GAFB NY 13441-4505, Phone: (315) 330-2762, DSN 587-2762)

The REE/ProSLCSE integration will provide a means for the specifications developed using advanced requirements engineering technology to be directly input to the process established and controlled by the ProSLCSE Environment, to track requirements throughout the remainder of the life cycle, and to provide complete requirements traceability. The integration will make use of both expert systems technology and existing object oriented database technology to determine and transfer the requirements from the Requirements Engineering Environment database to the ProSLCSE database. (Rome Laboratory Point-of-Contact: Ms. Elizabeth S. Kean, RL/C3CB, 525 Brooks Rd., GAFB NY 13441-4505, Phone: (315) 330-2762, DSN 587-2762)

3. SOFTWARE AND SYSTEM QUALITY

The Software and System Quality area provides technology to specify, measure, and assess the quality of the software product. If the process is instituted and managed as described above in the ProSLCSE, then the products should be more correct and reliable. The automated assessment of product quality enables the process to be measured and adjusted accordingly for optimum use of scarce resources. A framework and an automated tool called the QUality Evaluation System (QUES) (see Figure 3) has been developed at the Rome Laboratory for quantitatively measuring the quality of virtually every product of the software life cycle, from the requirements specification to the delivered software and documentation. Nippon Electric Company has already used the technology to achieve a net 25% increase in productivity for software development, and a decrease in of 51% in first year maintenance costs. The Rome Laboratory has demonstrated the feasibility of expert systems to assist in the selection and tailoring of these metrics in command and control, intelligence, and space applications. Software reliability/test integration techniques have been developed which combine software testing techniques such as path testing, symbolic execution, and mutation analysis with reliability assessment. The results of this effort will take the form of a guidebook. A modest effort for software quality methodology enhancements is examining the theoretical aspects of software quality metrics and software quality effects for advanced architectures such as parallel and

highly concurrent processing effort to address the integration and exploitation of software quality specification and assessment technology.

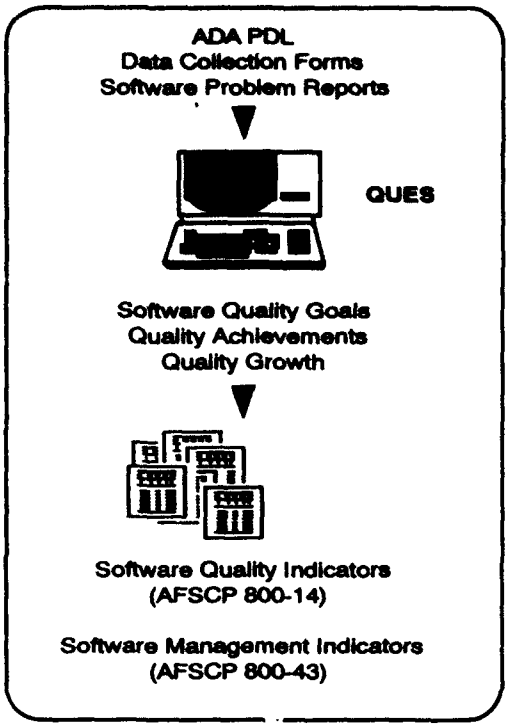


Figure 3
Quality Evaluation System (QUES)

A Cooperative Research and Development Agreement (CRDA) with industry is a joint venture to validate software quality technology, to establish benchmarks and baselines for comparative analysis, and to transition automated software quality technology from the laboratory to the field. The ongoing and planned activities by the members of the CRDA called the Software Quality Consortium are providing the means for both Air Force and industrial partners to acquire automated software quality technology, validate this technology on real-world problems, and to transition software quality tools and technology so that it becomes a part of the process of producing systems. The QUES tool is being used by the consortium to evaluate software development products and provide an assessment against specified software quality goals. (Rome Laboratory Point-of-Contact: Mr. Andrew J. Chruscicki, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-4476, DSN 587-4476)

4. SOFTWARE FOR HIGH PERFORMANCE COMPUTERS

The Rome Laboratory program in the area of Software Technology for High Performance computing is directed at the development of software engineering technology to cope with complex systems consisting of a mix of sequential and highly parallel computing equipments. Current investigations have produced reports which describe shortfalls in software high performance computer architectures. In addition, ongoing work is focused on the development of a Parallel Evaluation and Experimentation Platform (PEEP) (see Figure 4) which will enable researchers and developers alike to formulate new approaches to algorithm and software production which takes advantage of these performance multiplying computers. Areas to be addressed include the impact of parallel architectures on existing software design processes, identification of parallel processing tools and techniques to support software development for high performance computer architectures, understanding and isolation of target machine dependencies, tradeoffs between portability and efficiency, and the effect of program language selection on the software design process.

A Cooperative Research and Development Agreement (CRDA) in Parallel Software Engineering combines Air Force and industry resources to solve key problems faced by the use of highly parallel computers and the software that is run on these machines, often in a heterogeneous environment consisting of both sequential and parallel processors. The period 1995-2000 will be characterized by distributed computing among heterogeneous computers, many of which may be high performance computers.

Planned work in this area involves the specification of a virtual machine interface layer for interaction between parallel software tools such as those found on the PEEP and candidate architectures that may be selected for implementation. Portability will be a key factor in such a machine, new programming models will be considered, and the machine will cover a wide range of high performance computers. (Rome Laboratory Point-of-Contact: Mr. Paul M. Engelhart, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-4063, DSN 587-4063)

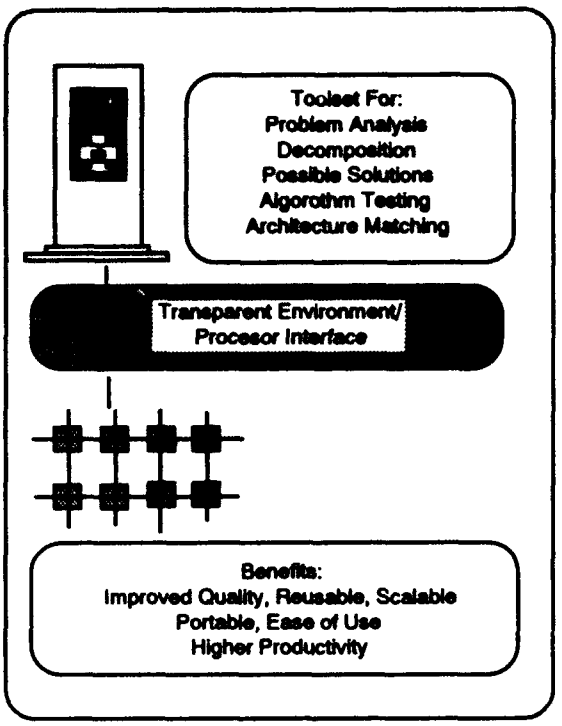


Figure 4
Parallel Evaluation and Experimintation Platform (PEEP)

An architecture independent parallel design tool is planned which will provide a means to design heterogeneous systems consisting of both sequential and parallel computing equipments. The tool will extend current design strategies for sequential tool building, develop utilities for aid in design understanding, and will demonstrate an advanced human computer interface for ease of use and tool applicability. (Rome Laboratory Point-of-Contact: Ms. Milissa M. Benincasa, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-7650, DSN 587-7650)

In future efforts software issues that must be addressed to realize the performance benefits of high performance computing include improving algorithm science, parallel languages technology and software development environments to help integrate computers of various designs. Computation strategies will be needed that can work with a variety of architectures (e.g., shared vs distributed memories) to obtain economy of use through algorithm and software reuse, to permit effective testing and to make available common high level packages such as fourth-generation languages which will be graphic-based, nonprocedural and end-user oriented. (Rome Laboratory Point-of-Contact: Mr. Joseph P. Cavano, RL/C3CB, GAFB NY 13441-5700, Phone: (315) 330-4063, DSN 587-4063)

REFERENCES

Boehm, B. W. "Software Engineering"; TRW-SS-76-08, TRW Defense Systems Group, Redondo Beach CA (October 1976).

Boehm, B. W. "Software Engineering Economics"; Prentice-Hall, Inc. New York NY (1981).

Burns, C., "Parallel Proto - A Software Requirements Specification, Analysis and Validation Tool, Proceedings AIAA Computing in Aerospace 8, Baltimore, MD (October 1991)

DiNitto, S., "Rome Laboratory"; Cross Talk, The Journal of Defense Software Engineering, Number 34, June/July 1992.

Green, C. et al.; "Report on a Knowledge-Based Software Assistant"; RADC-TR-83-195, Rome Air Development Center, Griffiss AFB, NY (August 1983).

Harris, D. and Czuchry, A.; "Knowledge Based Requirements Assistant"; RADC-TR-88-205, Vols. I & II Rome Air Development Center, Griffiss AFB, NY (October 1988).

Milligan, J.R. "The Process-Oriented Software Life Cycle Support Environment (ProSLCSE) "SEE" It Today (Tomorrow May Be Too Late)", Proceedings 4th International Conference on Strategic Systems, Huntsville, AL (March 1992)

Pease, D, "Parallel Computing Systems", RL-TR-92-131, Rome Laboratory, Griffiss AFB, NY (June 1992)

Rzepka, W. E. "A Requirements Engineering Testbed: Concept, Status and First Results"; Proceedings 22nd Hawaii International Conference on System Sciences", Vol. II, Kailua-Kona, HI (January 1989).

Strelich, T. "Software Life Cycle Support Environment" RADC-TR-89-385, Rome Air Development Center (February 1990).

Yau, S.S. et al; "A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems", Proceedings 16th Annual International Computer Software and Applicatiuons Conference, Chicago, IL (September 1992)

Discussion

Question R. SZYMANSKI

How will you make your current environment development efforts more successful than previous US DoD environment development efforts?

Reply

The approach we are using is to build a framework for the user interface and database and allow project managers to define off-the-shelf or their own internal tools to be incorporated within SLCSE. The key is for the data to be maintained throughout the lifecycle. Where we are building specific tools, we are using off-the-shelf tecnology and encouraging the developpers to commercialize the tools.

Question Dr GRANDI

You addressed the major area of SW re-use. Is the focus of your investigation on code re-use or is your approach more general and addresses also specification and design re-use? In this 2nd case, what is your strategic approach for identifying re-usable specification and design?

Reply

Each of the areas addresses re-use from their perspective. However, the system/software environments area is looking at re-use in general. They are addressing certification of specification, design, code, etc.

Question D. NAIRN

What is the relationship of your work to CAIS, APSE, european PCTE environment standardization programs?

Reply

The pro-SLCSE program is attending and monitoring all of the standards activities. In particular, pro-SLCSE is CALS compliant and supports POSIX. PCTE is closely monitored.

A Common Ada Run-Time System For Avionics Software

Clive L. Benjamin
Marc J. Pitarys
Wright Laboratory (WL/AAAF-3)
Building 635, 2185 Avionics Circle, Wright-Patterson AFB,
Ohio 45433-7301, USA

Eliezer N. Solomon
Steve Sedrel
Westinghouse Electronic Systems Group P.O. Box 746, MS 432,
Baltimore, Maryland 21203-0746, USA

SUMMARY

The United States Air Force (USAF) requires the use of the Ada programming language in the development of new weapon system software. Each Ada compilation system uses a Run-Time System (RTS) for executive services such as tasking, memory management, and system initialization. Implementing and using custom RTS services in each software development activity inhibits avionics software reuse and portability. In addition, the USAF must support many Ada compilers over the operational life of the weapon system. Finally, extensive testing and knowledge is required for each RTS.

In 1990 the USAF began defining a Common Ada Run-Time System (CARTS) for real-time avionics applications. A contract was awarded to Westinghouse Electric Corporation (WEC) with subcontracts issued to compiler vendors DDC-I, Inc. and Tartan, Inc. A specification for the CARTS was completed in 1991 and coding of selected CARTS features was undertaken. The specification defined common interfaces between the application software and the RTS, and between the Ada compiler and the RTS. Incremental coding of the CARTS prototype is being done by DDC-I for the MIPS R3000 processor, and by Tartan for the Intel 80960 MC processor. Several prototypes have been developed and tested. This paper covers the significant CARTS features and services offered to avionics software engineers. The paper provides the results of performance testing of the CARTS features. Finally, this paper provides information on the appropriate use of the CARTS by avionics software engineers.

INTRODUCTION

The CARTS effort seeks to address the issue of portability and maintainability raised by the use of custom run-time systems across different processors and compilers. It further seeks to demonstrate the feasibility of a common Ada run-time system. The CARTS is aimed at different compilers and targets. A complete implementation of the CARTS would allow for a seamless port of application software from one target to another, provided that the application utilized both non-compiler-specific and

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

non-target-specific code with CARTS system calls.

The prime contractor on the effort is Westinghouse Electric Corporation (WEC) and the subcontractors are DDC-I Inc. and Tartan Inc. The CARTS is being developed on a VAX/VMS based DDC-I cross-compiler for the MIPS R3000, and on a VAX/VMS based Tartan cross-compiler for the Intel 80960 MC.

CARTS is meant to be the center of a complete Run-Time System (RTS). The RTS consists of the Common Ada Run-Time System (CARTS), the Compiler Specific Run-Time System (CSRTS), and the Application Specific Run-Time System (ASRTS). The CARTS is the portion of the RTS where the interfaces and implementation are common across different Ada compilers for the same hardware environment.

The Compiler Specific Run-Time System (CSRTS) contains those portions of the RTS that may need to vary from one Ada compiler to another, but can be common across different applications using the same compiler.

The Application Specific Run-Time System (ASRTS) contains those portions of the RTS that for a given compiler and CARTS implementation may need to vary from one application to another.

CARTS is aimed primarily at the real-time Ada avionics community and seeks to address its needs. In so doing, CARTS builds upon the considerable work done by industry groups and by other members of the Ada community in the area of run-time systems. One of these organizations is the ACM SIGAda, Ada Run-Time Environment Working Group (ARTEWG). The ARTEWG sponsored efforts resulting in the Model Run-Time System Interface for Ada (MRTSI) and the Catalog of Interface Features and Options (CIFO) documents. These two documents form the basis for a major portion of the CARTS Software Requirements Specification (SRS) [1]. These documents, however, were not the only ones researched for the CARTS SRS.

Specific requirements had also been identified and documented by other members of the Ada community: the Joint Integrated Avionics Working Group (JIAWG) proposed requirements for a common Ada run-time system; ExTRA (Extensions Temps Real Ada) defined an interface which was developed under the auspices of Aerospatiale, the French Government Aerospace Agency; and, the Fourth International Workshop on Real-Time Ada Issues (RTAW4) reviewed Draft 2.0 of the Ada 9X Revision Requirements document. In addition to this, input and feedback on the features that ought to be included in the CARTS, was solicited from internal user and client communities of Westinghouse Electric Corporation (WEC), the two subcontractors, and the USAF.

CARTS IMPLEMENTATION DETAILS

As mentioned in the Summary, the CARTS Software Requirements Specification (SRS) was completed in 1991. However, due to funding constraints, only a portion of the entire SRS, represented by the selected features, was scheduled for the Design and Implementation of the Prototypes. The focus of the effort was directed to those CARTS features that would be the most useful, and hence have the greatest payoffs, from the perspectives of real-time avionics software engineers. These features, selected and prioritized by the team, constitute the focus of the discussion in this paper.

Portions of the CARTS have been implemented in three incremental prototypes as of the writing of this paper. Currently, a revision of the third prototype is being implemented and tested. The testing of CARTS is being carried out by the prime contractor, Westinghouse Electric Corporation (WEC). The testing falls into two

categories: CARTS-feature testing to assure compliance; and, performance testing to evaluate efficiency. CARTS will be discussed in more detail in the Testing section of the paper.

CARTS FEATURES

As indicated above, selected CARTS features, identified as being of high priority, and having been the subject of considerable discussion and debate, were proposed for implementation in the CARTS prototypes. These more salient features are described in the ensuing sections.

Task Scheduling

Task Scheduling has been identified as the feature of highest priority by the application development community. The operations that support this feature are contained in the package RTS_Primitives. One of the operations is procedure Suspend_Self which enables the calling task to suspend itself. This procedure has a parameter of type Suspension_ID. The type Suspension_ID is an integer type that is RTS-defined, and it must have a minimum of 254 values. These values identify the reason(s) for the suspension of a task. A second operation is procedure Resume_Task. This procedure causes the state of the "Suspended" task to be reset. This procedure has two parameters. One is of type Task_ID and the other is of type Suspension_ID. The type Task_ID will be defined in a later section. The third operation is procedure Yield. This procedure causes the task to yield its physical processor to a waiting task of equal priority. These procedures, in addition to those described in the following section, accomodate Task Scheduling. Another feature contained in package RTS_Primitives, is Asynchronous Transfer of Control, which will be discussed in a later section.

Dynamic Priorities

Dynamic Priority refers to the ability of a task's priority to be changed dynamically. Dynamic Priority is extremely important in fault tolerant software. This feature is also included in the package RTS_Primitives. The ability to change the priority of a task is provided by two operations supported by the procedure Set_Base_Priority. One Set_Base_Priority procedure, with two parameters of types Task_ID and System.Priority, allows the priority of a single task, specified by the Task_ID parameter, to be changed. The other Set_Base_Priority procedure, with a single parameter of type System.Priority, sets the priority of the calling task to the value of the parameter. Similarly, the ability to enquire about the base priority of a task is provided by two operations supported by the function Base_Priority. One Base_Priority function, with a single parameter of type Task_ID, returns the base priority of the task indicated by that parameter. The other Base_Priority function returns the base priority of the calling task

Task Identifiers

Task Identifiers have been identified by the authors of CIFO and of the Fourth International Workshop on Real-Time Ada Issues (RTAW4) [2] as a feature of importance. Task Identifiers are a storable type used to uniquely identify a specific task to the RTS. This requirement is met by the package RTS_Task_ID. This package defines a type Task_ID and some basic functions on Task IDs. The function Same_Task checks whether the two parameters, of type Task_ID, identify the same task: the function returns a value of type Boolean. The function Self returns the Task_ID of the calling task. Task Identifiers, as mentioned in an earlier section, are used as a building block for other operations. They are used to implement Abort_Tasks, a procedure contained in the package RTS_Task_Stages. This procedure is the only feature that has been implemented in that package. The procedure

implements the Ada Abort statement. The parameter of the procedure is a list of valid Task_IDs.

Interrupt Handling

Interrupt handling is another of the high priority features. This feature is also contained in the package RTS_Primitives. The package contains support for the use of procedures as interrupt handlers. The types of interrupts supported are specified here.

Hardware Interrupts

Hardware interrupts are specific to a physical processor. The characteristics of the physical processor define the behavior of a hardware interrupt. The hardware interrupt handler procedure may neither propagate an exception, nor cause a transfer of control directly in the interrupted thread of control. Furthermore, it is globally bound to the corresponding hardware interrupt.

Traps

Traps are internal signals which are the result of an anomalous execution state in a particular task. A trap may cause an exception to be raised in the corresponding task. The binding of a trap to a trap handler procedure is accomplished by the task executing the Attach_Interrupt_Handler routine.

Virtual Interrupts

Virtual Interrupts can be used by one task to effect an Asynchronous Transfer of Control within another, target, task. A virtual interrupt handler procedure is associated with the target task which attaches the virtual interrupt to the handler procedure. This procedure is executed when the virtual interrupt is delivered to the target task as the result of a call on the Interrupt_Task operation. These Virtual Interrupts are supported by several operations, described herein. The function Interrupt_Priority, which returns the priority associated with the specified interrupt, has a single parameter of type Machine_Specifics.Interrupt_ID. Attach_Interrupt_Handler, binds a handler procedure to the specified interrupt, and has three parameters of the following types: Machine_Specifics.Interrupt_ID; System.Priority; and, System.Address. The procedure Detach_Interrupt_Handler detaches the specified handler from the specified interrupt and restores the system default handler. Interrupt_Task delivers the specified virtual interrupt to the specified task. A more detailed discussion of interrupt support in CARTS is the subject of another paper entitled *"Real and Virtual Interrupt Support: The Mapping Of A CARTS Feature To Two Different Architectures"* [3], and is being presented at Ada Europe '93.

Clocks and Delays The Fourth International Workshop on Real-Time Ada Issues (RTAW4) [2] identified a requirement for a non-adjustable monotonic clock which is used for delays and for periodic execution. This requirement is the rationale for most of the package RTS_Clock. The function Clock returns a value that is monotonically increasing over time. The package also contains operations Delay_Self and Delay_Until, and ancillary arithmetic functions. Delay_Self is an operation that allows the calling task to block itself until a time of D * seconds_per_time has elapsed; it takes a parameter of type Fine_Time. Delay_Until is a procedure that causes the calling task to be suspended until the Clock has reached a specified time T; it takes a parameter of type Time. Finally, the arithmetic functions add and subtract parameters of type Time and Fine_Time.

Asynchronous Entry Calls

Asynchronous Entry Calls (AEC), an important CIFO requirement that has been fastidiously supported by the ARTEWG, was implemented in one of the earlier CARTS Prototypes. Whereas the

Ada tasking model supports only synchronous communication via entry calls, the AEC supports asynchronous communication via entry calls. This mechanism allows the application developer to enqueue an entry call to a task without waiting for that task to accept the call. In the CARTS, the package RTS_Asynchronous_Calls contains the procedure Call_Asynchronous which has two parameters, one of type Agent_Collection_ID, and the other of type Parameter_Block. The Agent_Collection_ID type is used to identify the collection of system resources allocated to execute the asynchronous entry call. These system resources are called Agents. The Parameter_Block type is used to represent the block of parameter data that is transmitted by the asynchronous entry call. The collection of Agents is created by the function New_Agent_Collection which has four parameters: Acceptor, of type Task_ID; E, of type Entry_Index; Number, of type Positive; and, Length of type Positive. The Task_ID and Entry_Index represent, respectively, the task and entry calls that Agents shall use to queue calls. The first positive parameter refers to the number of Agents involved, while the second positive parameter refers to the size of the Parameer_Block. A more detailed discussion of the CARTS AEC support is the subject of another paper entitled *"The Implementation Of Asynchronous Entry Calls On Two Different Architectures"* [4], and is being presented at the National Aerospace Conference (NAECON '93).

TESTING

While the primary responsibility for the design and implementation of the CARTS features into the compilers was the domain of the compiler vendors, DDC-I and Tartan, the primary responsibility for the testing was the domain of Westinghouse Electric Corporation (WEC). Although revisions to the third and final Prototype are being implemented as of the writing of this paper, some preliminary testing has already been conducted on each of the two final CARTS-compliant Prototypes, for the two target architectures. The testing can be divided into two categories as mentioned above. One category is the testing for compliance with the CARTS SRS, and the other category is the testing to assess the run-time performance of the implementation of the CARTS features.

CARTS-specific tests were developed to test compliance of the implementation of the CARTS features. Almost all of the CARTS features that have been implemented in both compilers, have been tested for compliance. This has been accomplished by code inspection and by conducting the CARTS-feature tests.

The performance testing consists of a subset of the Joint Intergrated Avionics Working Group (JIAWG), Performance Issues Working Group (PIWG), and Ada Compiler Evaluation Capability (ACEC) benchmarks. The performance testing was done to assess whether the use of the various CARTS features that were implemented was accompanied by any significant overhead and run-time degradation. The implementation of CARTS into the Tartan baseline compiler for the 80960 MC caused no degradation in run-time performance of the executable code. As a matter of fact, the Tartan 80960 MC compiler showed the same execution times for all three prototypes. The DDC-I compiler, on the other hand, showed significant improvement in the first two prototypes. The difference in the execution speeds can be attributed to other improvements made in the compiler from DDC-I.

In addition to this type of unit testing, WEC has developed a single application that takes advantage of several CARTS features, all designed into the application. A discussion of this application is presented herein.

CARTS APPLICATION

An application, the Distributed_Mail_Router (DMR), has been developed by Westinghouse Electric Corporation (WEC), to demonstrate the various CARTS features which have been implemented to date. The following discussion describes the application and the CARTS-specific support used to facilitate its design and implementation.

Description of the CARTS Application

The Distributed_Mail_Router (DMR) is a distributed system employing a mail-box mechanism for inter-task communication. A logical system view is presented in Figure 1. The different application tasks communicate with each other via a mail-box scheme which is transparent to the actual physical distribution of the various application tasks. A typical physical task distribution is shown in Figure 2. Although Figure 2 illustrates three CPUs per chassis, and three chassis, the system is obviously extensible to N CPUs as shown in Figure 3. This particular configuration was chosen with a view to facilitating interactive debugging. Cooperative scheduling between the application tasks is employed in order to manage the CPU resource on each module of the system. The only exception to the cooperative scheduling scheme is the occurrence of an interrupt (either hardware or virtual).

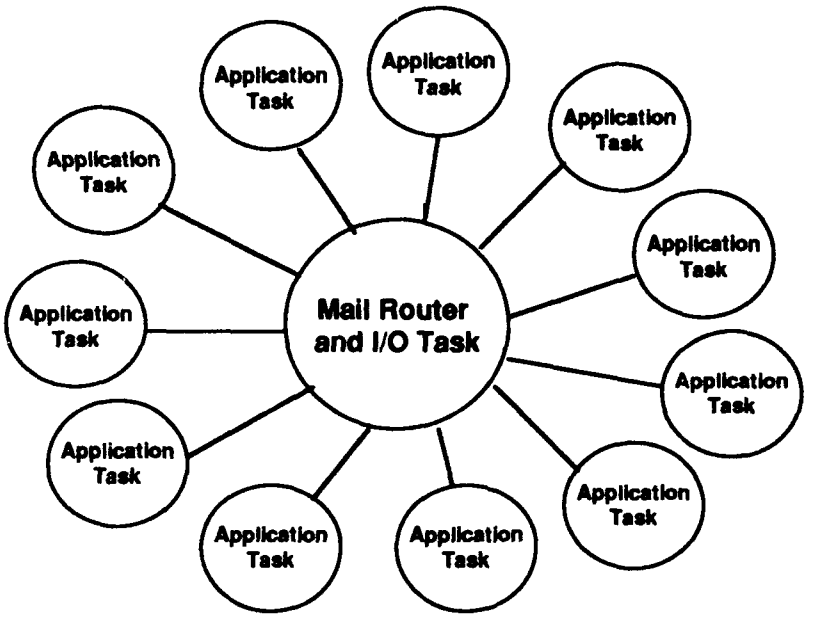


Figure 1. Logical View Of The System.

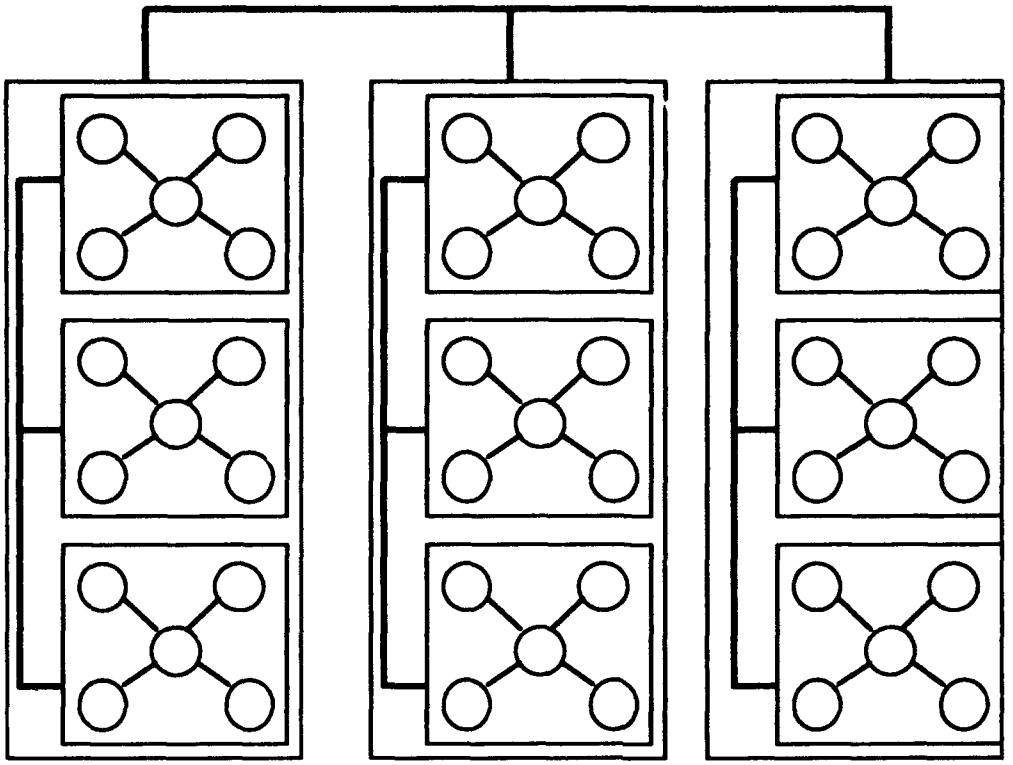


Figure 2. Physical View Of The System.

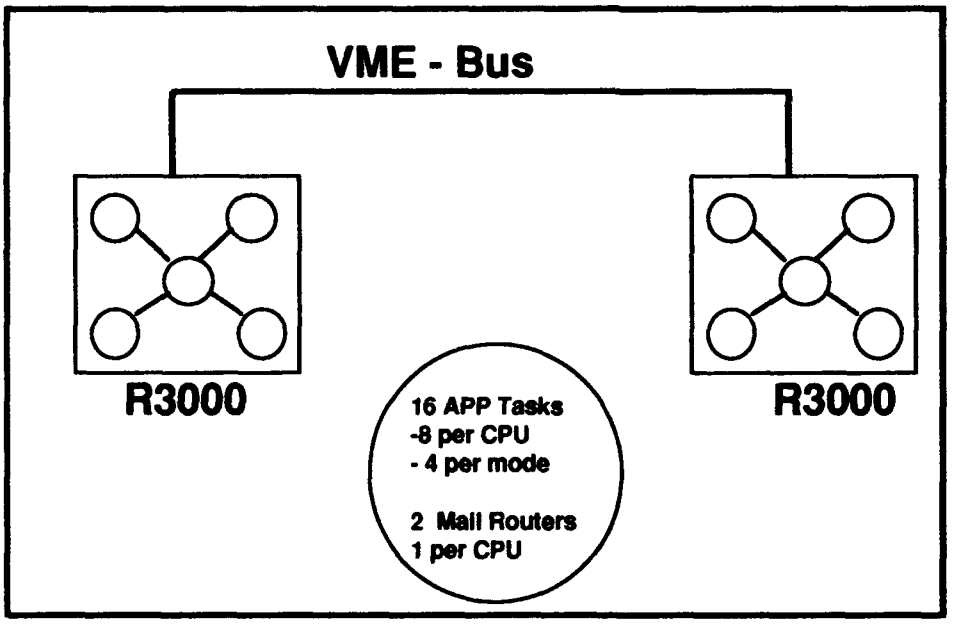


Figure 3. Actual View Of The System.

When Ada '83 is used to realize a mail-boxing scheme, mail-boxes are typically implemented as buffer tasks associated with event flags which provide a means for a task to release the processor resource while it is "waiting" on an event to occur, such as the receipt of mail or an I/O event. Our approach to the implementation of the mail-boxes uses the Asynchronous Entry Calls (AEC) feature of the CARTS to provide the same functionality as an Ada '83 approach. The Asynchronous Entry Calls feature of the CARTS allows the user to designate an entry(s) as an asynchronous entry(s). As a result of this action a queue is associated with the designated entry which allows the caller of the entry to asynchronously queue a data structure which contains all the information the accepting task requires to execute the accept block at some "later time".

In the CARTS implementation, the interface to the Asynchronous Entry Calls functionality is provided by the package RTS_Asynchronous_Calls via subprograms and two data types. The private type Agent_Collection_ID, and the function New_Agent_Collection, are used to allocate collections of memory, which are used for the entry queues of the asynchronous entries. A call to the function New_Agent_Collection returns a value of type Agent_Collection_ID which is used as a handle to identify that particular collection of memory in a call to the Call_Asynchronous procedure. The New_Agent_Collection function also determines the number of asynchronous entry calls that may be queued at one time, and the size of the parameter block (the other data type defined in the package) which is to be used for passing parameters to the accept statement. In the DMR application, the number of entries to be queued represents the number of mail messages for that mail-box, and the size of the parameter block is the size of the actual message. Two entries per task are used as the mail-boxes, one being a Low_Priority mail-box, and the other being a High_Priority mail-box. The Virtual Interrupt handler then transfers a mail-box message to the high or low priority mail-box using the Call_Asynchronous procedure of the RTS_Asynchronous_Calls package. The application task manages its mail-box servicing scheme by utilizing the "accept" statement, and the "count" attribute for the entries designated as the mail-boxes for that task.

The mail-box servicing scheme is implemented on top of the cooperative scheduling philosophy. Each of the application tasks contains a main processing loop which is executed after the initialization preamble. The algorithm for the main loop is presented here.

```
if (High_Priority_Mailbox'COUNT > 0) then
   MAILBOX_STATUS := HAVE_HIGH_PRI_MAIL;
elsif (Low_Priority_Mailbox'COUNT > 0) then
   MAILBOX_STATUS := HAVE_LOW_PRI_MAIL;
else MAILBOX_STATUS := NO_MAIL;
end if;
loop
  case MAILBOX_STATUS is
    when NO_MAIL =>
      select
        accept High_Priority_Mailbox do
          copy message to local variable
        end High_Priority_Mailbox;
      or
        accept Low_Priority_Mailbox do
          copy message to local variable
        end Low_Priority_Mailbox;
      end select;
    when HAVE_LOW_PRI_MAIL =>
      Yield;  -- Allow another task to execute
      accept Low_Priority_Mailbox do
        copy message to local variable
      end Low_Priority_Mailbox;
    when HAVE_HIGH_PRI_MAIL =>
      accept High_Priority_Mailbox do
        copy message to local variable
      end High_Priority_Mailbox;
  end case;

    ... process mail message ...

  if (High_Priority_Mailbox'COUNT > 0) then
     MAILBOX_STATUS := HAVE_HIGH_PRI_MAIL;
  elsif (Low_Priority_Mailbox'COUNT > 0) then
     MAILBOX_STATUS := HAVE_LOW_PRI_MAIL;
  else
     MAILBOX_STATUS := NO_MAIL;
  end if;
end loop; -- Main processing loop
```

The invocation of the Yield procedure is a dispatching point that causes the task which called it, to be placed at the end of the dispatch chain for the tasks of equal priority.

In addition to modeling distributed tasking, the system also models a mode switching application. In this sense, the system can be viewed as having four distinct modes of operation, and each mode is composed of a set of tasks that implements the required functionality of the mode. The Virtual Interrupt capabilities discussed in this paper, along with the CARTS RTS_Primitives subprograms, are used in the system to allow rapid Asynchronous Transfer of Control (ATC) between tasks in order to effect rapid reconfiguration of the application tasks during mode change.

The rest of this discussion is oriented towards the interrupt aspects of the implementation. Each CPU in the system has a "routing table" that contains the information required by the mail routing subsystem, and the mode switching subsystem. Messages which are passed between CPUs contain a header with information which distinguish the message either as a mail-box message, or as a mode change message. Depending on the type of the message, additional fields of the header provide the parameter data needed by the mail routing and mode switching algorithms. In the case of a mail-box message, a high priority interrupt service task (Virtual_Interrupt, explained below) receives the message via an Asynchronous Entry Call, and it forwards it via another Asynchronous Entry Call to the appropriate task. If the message is a mode change, then the hardware interrupt handling procedure calls the Virtual Interrupt procedure associated with the interrupt service task, and the handler effects the mode change for all mode-specific application tasks resident in each CPU of the system.

Before going any further with the details of the Virtual Interrupts, it is necessary to briefly explain the priority scheme used by the system. The highest priority is a group of priorities called the Hardware_Interrupt range. The next priority is a single priority called the Critical_Region priority that is represented by Hardware_Interrupt'first - 1. Immediately below the Critical_Region priority is another range of priorities called the Virtual_Interrupt range. Below the Virtual_Interrupt range is the Asynchronous_Transfer_Of_Control trigger range. Finally, at the bottom is the Application_Task range. These map into the CARTS and Ada9X priority levels as follows: the Hardware_Interrupt range is the same as the System.Interrupt_Priority range, and all others fall into the System.Priority range. The Critical_Region priority is used to implement mutual exclusion in the regions below the Hardware_Interrupt priorities by calling a procedure in package RTS_Primitives which elevates the caller's Base_Priority to this level and masks off all hardware interrupts. When the critical region is exited, the Base_Priority is returned to its prior value and the interrupt masks are restored to their previous values. The elaboration of each application task triggers several actions: it allocates its high and low priority mail-boxes; it fills in the "routing table" with the necessary configuration information; it sets up its Virtual_Interrupt handler procedure; and, it then suspends itself by a call to the procedure RTS_Primitives.Suspend_Self. The Virtual_Interrupt handler is the same for each application task. An example of this is provided below.

```
procedure Application_Task_Virtual_IH is
  Mode : Global_Types.Mode_Type;
--      Mode_1 ... Mode_4
begin
   Determine the current mode via the application
   task's  Current_Mode task Attribute_ID

   Determine if the application task should be active
   in the current mode of operation by checking the
   "routing table"

   If (not Active in this mode) then
    call RTS_Primitives.Suspend_Self
```

```
      call RTS_Primitives.Set_Base_Priority to reset the
        task's  priority to its "normal"  value in the
        "routing table"

    else
      call  RTS_Primitives.Set_Base_Priority to reset the
      task"s  priority to its "normal" value in the
      "routing table"
    end if;
end Application_Task_Virtual_IH ;
```

A mode change is executed when the Virtual_Interrupt handler for the interrupt service task becomes active.

```
procedure Mode_Change_Virtual_IH is
begin
   Enter Critical_Region

   Determine the new mode by accessing the
   Current_Mode task  attribute of the task to which
   it is bound.  This was set by the hardware
   interrupt  handler procedure  prior to
   triggering this Virtual Interrupt handler.

   Set the Current_Mode task attribute of all the
   application   tasks using the data in the "routing
   table"

   Exit Critical_Region

   Queue an Asynchronous Entry Call of the interrupt
   service task to acknowledge the mode change to
   the  other CPU

   Give all currently active application  tasks
   "pending"   Virtual_Interrupts by calling the
   RTS_Primitives.Interrupt_Task procedure for each
   Task_ID   that is designated as Active for the
   previous mode

   Call the RTS_Primitives.Resume_Task for each
   Task_ID   designated in the "routing table" as
   Active for the new mode

   Call the RTS_Primitives.Set_Base_Priority
   procedure  for each  application task with a
   pending Virtual_Interrupt, setting them to the
   Asynchronous_Transfer_Of_Control trigger level in
   order to effect an Asynchronous Transfer of
   Control  to the application tasks as soon as this
   procedure  and any pending hardware interrupts
   are complete

end Mode_Change_Virtual_IH;
```

CONCLUSION

The development of the CARTS SRS is a manifestation of the direction to standardize the interfaces for Ada RTS's. The inherent program constraints, such as funding and schedule, dictated that only portions of the entire SRS, would be designed, implemented, tested, and demonstrated. These portions of the SRS are represented by the features selected as a result of the feedback obtained from a cross-section of the Ada community: real-time embedded systems developers; client communities of the compiler subcontractors; and, the USAF. This feedback indicated a far greater desire for run-time systems to provide a real-time embedded system application developer with support for additional run-time services that could be directly accessed by the application. The focus of the effort was thus directed to those CARTS features that would be the most useful, and hence have the greatest payoffs, from the perspectives of real-time avionics software engineers. As is evident, these features, selected and prioritized by the team, and providing such support for access to the low-level features of the run-time system, constituted the focus of the CARTS work done to date.

CARTS is a good baseline to assess the concept of a common Ada run-time system. Testing clearly shows that application software, that would otherwise be compiler- and/or target-specific based on specific run-time calls, can be ported across CARTS-compliant compilers/ run-time systems.

ACKNOWLEDGMENT

Special mention must be made of Theodore P. Baker, Ph.D., of Florida State University, Tallahassee, Florida, USA. Dr. Baker is the primary author of the Software Requirements Specification for the Common Ada Run-Time System (CARTS). The CARTS project would not have been successful without his assistance and guidance.

REFERENCES

[1]. Software Requirements Specification for the Common Ada Run-Time System (CARTS). Version 1.0, 1991.

[2]. Proceedings, Fourth International Workshop on Real-Time Ada Issues (RTAW4), Pitlochry, Scotland, 1990.

[3]. R. Mancusi, J. Tokar, M. Rabinowitz, E. Solomon, M. Pitarys, C. Benjamin. "Real and Virtual Interrupt Support: The Mapping Of A CARTS Feature To Two Different Architectures", to be presented at Ada Europe '93.

[4]. A. Fergany, L. Szewerenko, M. Rabinowitz, E. Solomon, M. Pitarys, C. Benjamin. "The Implementation Of Asynchronous Entry Calls On Two Different Architectures", to be presented at the National Aerospace Conference (NAECON '93).

Discussion

Question W. MALA

Will your "common runtime system" be fully compatible with Ada-9X?

Reply

The common runtime system has been developped based on Ada 83. How far the "common runtime system" will be compatible with Ada-9X cannot be answered yet, because the definition of Ada-9X has been completed a few month ago. Further investigation will be necessary.

Question P.C. BROWN

Is there any intention to put the standard RTS forward as a public domain standard?

Reply

At this time, there are no plans to the Common Ada Runtime System as a public standard.

Ada Run Time System Certification for Avionics Applications

Jacques Brygier - Marc Richard-Foy

Alsys
29 Avenue Lucien-René Duchesne
78170 La Celle Saint Cloud
France

Abstract. The certification procedures apply to a full equipment including both hardware and software components. The issue is that the equipment supplier must integrate various components coming from separate sources. In particular, the Ada Run Time System is embedded in the equipment as any other application component. This leads to two major requirements:

a. the Ada Run Time System must be a glass box
b. unused run-time services must be eliminated from the embedded components.

The first requirement comes from the civil aviation procedures DO 178A [1] and the second is a consequence of the need to proof the system. This can lead to eliminate some unpredictible or unsafe Ada language features. The criticity of the system consists of three levels: **critical, essential** and **non essential**. The report ARINC 613 (from the Airlines Electronic Engineering Committee) surveys the Ada language and provides a list of features not to be used in avionics embedded software at least for the two first levels.

Two solutions are proposed:

1. The SMall Ada Run Time System (SMART) which meets such requirements for an Ada subset. This Run Time System does not support tasking, exception and dynamic memory allocation except for global objects or fixed size collections. We show how calls to this reduced Run Time System can be generated by the standard Ada compilation system.

2. The alternative Run Time System called C-SMART which is an approach used by Boeing with the cooperation of Alsys for the B777 project. C-SMART shares most of the SMART functionalities. Two major differences exist: it requires a devoted Ada compilation system and Alsys provides the end-user with the test plan of C-SMART which consists also of unitary tests set.

1 Introduction

Software now pervades almost every aspect of human endeavour. Our transport systems depend on software for both control of vehicles and the infrastructure. Our financial systems depend on software for the control of production. Our hospitals depend on software for the control of life-support systems. The use of software has grown over the last decade with the availability of low cost, high performance hardware. This growth has been almost surreptitious and it is only recently that society has realised that the safety of many human lives and much property now depends directly or indirectly upon the correctness of software.

The major attraction of software is its flexibility. However this very flexibility brings with it a greatly increased chance of error. There is now a strong awareness that positive measures are required in order to reduce the risks of errors in what has come to be called **Safety Critical Software**.

There is a consequential growing concern in all major industrial nations regarding the legal obligation of companies and their officers to ensure that systems do not violate safety requirements. Thus an officer of a company might be held personally liable for loss of life or property caused directly or indirectly by incorrect software installed in the company, sold by the company, or installed in a product sold by the company.

An extensive discussion of safety critical systems which are usually also real-time control systems will be found in reference [2].

2 The Avionics Example

The avionics industry has taken the lead in the development of safety critical systems. A modern commercial aeroplane contains a diverse combination of computers. These computers may control non-critical functions such as the entertainment systems or cabin lights, or safety critical systems such as engines, flaps, ailerons, or brakes.

Before an aeroplane can carry fare-paying passengers, it must undergo a thorough certification process to provide an acceptable level of confidence that it is safe to do so. The certification process starts early in the development of an aeroplane. Confidence in the safety of an aeroplane is built up with the aeroplane itself. Each component of the aeroplane is assigned a criticality level, depending on the effects its failure would have on the safety of the passengers. The confidence in each component must match the adverse effect that the component would have should it fail.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

As many of the components of an aeroplane are controlled by computer software, the safety of the components is critically dependent on the safety of the embedded software. The critical role that software plays in the safety of an aeroplane has forced the development of guidelines to help independent assessment of its safety.

The Radio Technical Commission for Aeronautics is an organisation in the United States with representation from Government, Airline, Airframe and component manufacturers. The RTCA published a document (Number RTCA/DO-178) in January 1982, called "Software Consideration in Airborne Systems and Equipment Certification". This document was later revised in co-ordination with the European Organisation for Civil Aviation Electronics (EUROCAE) and was published as DO-178A in March 1985. A further revision, DO-178B, is expected to be published in 1993. By the beginning of 1993, most new software certification efforts will be conducted under the guidelines of this new document.

These and other safety standards all require or recommend the use of best practices in all aspects of systems development. One area which is of key importance is the programming language used as the basis of the final installed system. The standards specify the use of a language which is well defined, has validated tools, enables modular programming, has strong checking properties and is clearly readable.

The conclusion is inevitable: of all the widely available languages, only Ada is an appropriate baseline for safety critical software.

3 The Software Development Process

All Certification Guidelines stress the importance of a process based on sound engineering practice. The steps to be taken in the development of the safety critical software must be well understood and documented before the software can be certified for safety critical applications. Rather than waiting until the software is fully developed and tested, it is wise to involve the certification authorities in the early planning stages.

3.1 Software Development Plan

To raise the confidence in safety critical software, the development stages used in its production must be understood. Software developed by a software engineer working alone does not instil the confidence required to flight certify a system. A preferred approach is to use a team which follows a controlled software engineering method. It is important that the method be used consistently on the project.

One of the first documents to be produced is the Software Development Plan. This plan must describe the development stages together with a description of the materials developed and their acceptance criteria. The software development plan must describe the production of the Requirements document, the Design document, the Software test plan and the Configuration plan.

3.2 Software Verification Plan

The Software Verification plan describes how the software is shown to be safe. It describes how the evidence for the assessment is collected and how it is presented. The verification of systems safety is done by review, testing and possibly format analysis. The plan must show how confidence is built up in the lower level software components and how the integration of these components satisfies the requirements of the system as a whole. Although thorough testing is always required, there is some debate over the use of formal verification methods at the source level.

Stylised mathematical equations which express the intent of a program may be mapped to source code. Tools which perform mathematical instructions can perform various checks which test the correctness of the set of equations. Formal verification of the source code alone cannot show that the software is safe.

Any tool used to verify the safety of an application is subject to the same level of verification as the application (if there is no further analysis done on the tool's output). Demonstrating the correctness of a program at the source code level does not guarantee that the generated code and the way is operates on the designated target processor configuration is safe.

4 Testing for Safety

Several kinds of testing strategies are required to achieve confidence in a safety critical system. "Black Box" testing checks that each function generates the expected results under all conditions that the function may experience. The goal of these tests is to check the behaviour of the function based on its observable effects. Each function must be tested with its typical data values, and also with its data values at the boundaries to check the extreme conditions which may be experienced.

"Glass Box" testing is a more stringent testing process. It involves analysing the structure of a function under test to ensure that all the elements of the function are required, all the elements are executed, and that all execution paths in the application are adequately covered. The tests must ensure that the program executes all conditions, and must also ensure that all conditions work correctly when evaluated to both true and false.

Multiple conditions tests are more difficult to formulate. They require tests which set all conditions to a known state, and then each individual condition is set and reset to ensure that setting and resetting individual conditions causes a desired effect. Programs may be transformed by assigning a multiple condition expression to a Boolean variable. This assignment would precede to conditional statement which would simply test the Boolean variable. Such a transformation does not reduce the testing

requirement. Each condition must be toggled and the result which is assigned to the Boolean variable must be checked against the branches of the conditional statement taken.

The tests and testing environment have to be designed to ensure that the tested software is as close to the final configuration as possible. If the testing environment is intrusive, the test results must describe the differences that can be expected between the tested and final product.

All the System and Software Requirements must be adequately covered by tests. All Derived Requirements, which are implicit, must also be adequately covered by tests. The derived requirements address features like initialisation of the stack, or set-up of heap addressing registers. To ensure that every requirement, and every byte of code is tested, a compliance matrix must be produced which records the relationship between documents, code, tests and test results.

5 Safe Ada Programming

Ada has properties which make it a natural choice for the development of safety critical systems.

- Ada is an ANSI and ISO standard. It is well defined and stable and thus provides a portable foundation for the development of supporting tools and libraries. The well-established validation mechanism gives trust in the general quality of Ada compilers.

- Ada supports Object Oriented Design. In particular there is strong support for abstraction and the reused of tested components.

- Ada has a legible style. This is important for the satisfactory execution of certification steps such as peer review and walkthroughs as well as later maintenance.

- Ada has a coherent modular construction. Separate compilation facilities enable the application to be written as a set of units, with interface specifications and the dependencies on their use clearly exposed. Ada compilers enforce these dependencies and ensure that if any interface specification is recompiled, then the corresponding unit that uses the interface must also be recompiled. Generally this enables the organised construction of a program from trusted components.

- Ada aids the detection of errors at an early stage of development. Strong typing facilities enable the user to construction programs where the way data is used depends on the way it was declared. Properly used this ensures that most errors are detected statically (that is by the compiler) and that many remaining errors are automatically detected at execution.

- Low-level features are provided through which the basic elements of the target hardware may be accessed in a logical manner. The address representation clause, enumeration representation clause, and unchecked conversions are some of the features which enable the program to be mapped to the target processor directly.

- Control over the visibility of types, operations and data provides a way of limiting the features which may be used by any program unit. For example, before the generic function **UNCHECKED_CONVERSION** can be used, it must be made visible by a **with** clause. This exposes the places that this potentially unsafe feature can be used, and allows special treatment and testing to ensure that the safety of the program as a whole is not compromised.

Ada is, however, a general purpose language and there are a number of Ada language features which should not be used in safety critical systems. A safety critical program must be totally bounded in time and memory used. This time taken to execute, and the amount of memory used by each element of the program must be determined and verified as part of the certification process. It is extremely difficult to determine the bounds of certain Ada constructs as they include call to the run-time system which may need to traverse run-time data structures which change as the program executes.

Tasking operations require calls to run-time routines which may scan various queues and analyse several data structures during the scheduling process. The time taken to perform these operations depends on the state of the tasking queues, which in turn depends on the state of the tasks at any given time. The time taken by the task operations cannot be determined unless all the possible states of the tasks can be determined at each of these run-time calls. Writing a test for each tasking operation under each of these combinations of task states is formidable. The current state of the art precludes such testing. Dynamic use of tasks is thus not recommended in safety critical systems.

The time and memory used during the elaboration of exception declarations and during the raising of an exception are predictable. Finding an exception handler once an exception has been raised involves searching through subprogram stack frames, or loading through tables which hold exception handler addresses. The time taken to locate the appropriate handler depends on the dynamic nesting of subprograms. Testing for all possible subprogram combinations at each point an exception can be raised presents a combinatorial explosion of states, as exceptions can be raised implicitly during program execution. Thus the used of dynamic exceptions is to be avoided in safety critical systems.

Use of heap storage presents a number of problems for certification. Memory for data to be placed on the heap can be claimed dynamically and released dynamically as well. The order in which the heap space is claimed depends on the use of the execution of the allocator. This

calls a run-time routine as and when one is required. Storage is deallocated, explicitly by the use of **UNCHECKED_DEALLOCATION** or implicitly, when an access type goes out of scope. The order of deallocation is not necessarily the reverse order of allocation. This fragments the free space in the heap. To minimise fragmentation, the run-time system typically uses algorithms to fit requests for space by searching for space availability. Various algorithms may be used: first fit, best fit and so on. As these searches are not deterministic, they are not permitted in safety critical systems. Although the heap could be used, its use must be restricted to a predictable set of operations where the time and memory used can be determined by analysis, and verified by testing.

We have thus seen that although Ada provides an excellent foundation for safety critical systems nevertheless certain features need to be avoided. In order to meet this requirement a number of *safe subsets* have been defined. One such system and its supporting tools is the SMART and C-SMART systems developed by Alsys.

6 The Run Time System

Ada programs implicitly require run-time system support during a program's execution. The run-time system must be provided in the program library which is used during the compilation of an application. The application developers have control and visibility over their own code, but the Ada run-time system is usually not visible to the user.

The Ada run-time system is written to satisfy the requirements of the Ada language, and the compiler whose output it must support. The design of the run-time system and the code generator are inextricably linked.

The run-time system is, of course, part of the delivered executable program and consequently for a safety critical application it must be subject to the same level of design and testing as the application code itself. Consequently, as part of the deliverables, full documentation and certification materials must be supplied not only for the application code but also for the run-time system actually used.

The normal run-time system for full Ada is not appropriate for a safety critical system (which uses only a subset of the language) because it contains code which uses techniques outside the certifiable regime. It is thus necessary to provide a run-time system appropriate to the level of subset being used.

7 SMART and C-SMART Systems

We have thus seen that although Ada provides an excellent foundation for safety critical systems nevertheless certain features need to be avoided. In order to meet this requirement a number of subsets have been defined. Alsys proposes two solutions, one, SMART, which can be compared to a "Glass Box" and the second, C-SMART, which can be compared to a "Black Box".

7.1 SMART

SMART has been designed to satisfy three general requirements:

- to have the smallest run-time code possible in the application
- to enable the use of a non-Ada specific real-time kernel
- to provide the basis of a safety critical solution

Minimal run-time

SMART is a run-time executive that is based on the principle of a "zero byte run-time", hence its name SMall Ada Run Time. It belongs to the ARTK (Alsys "Ada Real Time Kernel") environment, and provides a possible alternative to users who require a small Ada run-time.

The way in which SMART achieves a minimal Ada run-time is to minimise the run-time code in addition to the standard Alsys elimination of subprograms that are not used at link time. In minimising the Ada Run-Time System code, restrictions on the use of the Ada language are introduced. For example exception handling, tasking, and input/output are not supported.

SMART supports a restricted heap mechanism in which the services for allocating and deallocating objects are redirected to two user-provided routines ALLOC and FREE. No capability for pragma CONTROLLED or garbage collector is provided. Though this restricted model allows dynamic allocation / deallocation, it is mainly intended for static heap. A static heap is a dynamic memory area where objects are allocated at run time only once for ever for the whole program lifetime. This heap policy allows the declaration of unconstrained types or big objects without restricting too much the Ada language subset.

Therefore the SMART approach corresponds to a Subset of Ada whose restrictions are as follows:

- *Operation on discrete types*
  The attributes T'IMAGE, and T'VALUE are not allowed.

- *Array types*
  Constrained and unconstrained array types are generally allowed. However, unconstrained array types must be used with caution, as memory may be consumed by the heap and never deallocated.

Because type string is a special case of unconstrained array type, the restrictions described above also apply to string.

- *Discriminant constraint*

  Constrained and unconstrained records are generally allowed. However, as in the case for unconstrained array type, handling of unconstrained record types can lead to memory waste.

- *Allocators*

  Only objects of a collection without a 'STORAGE_SIZE clause or with a 'STORAGE_SIZE = 0, can be allocated. Declarations of access type on a collection with a 'STORAGE_SIZE clause are not allowed.

- *Tasks*

  Not supported

- *Exceptions*

  Ada exception handling is not supported

- *Unchecked Storage Deallocation*

  The restrictions on unchecked storage deallocation are those resulting from allocators

- *Predefined Language Attributes*

  The following attributes are not allowed:

  On open types:
  P'COUNT, P'CALLABLE,

  P'TERMINATED

  On integer types or subtypes: P'IMAGE, T'VALUE, P'WIDTH

  On enumeration types or subtypes: P'POS, P'WIDTH, P'SUCC, P'PRED

  On fixed point types: P'MANTISSA, P'LARGE, P'FOR

The SMART environment comes with a standard compilation system plus a specific option which enables to check that this Ada subset is met.

It is interesting to note that the Ada Run Time System (ARTS) subprograms of a SMART environment fall into four classes of services :

1. relay to user-provided processing (dynamic allocation / deallocation routines).

2. "null procedures" (i.e. begin null ; end ;). This kind of procedures are necessary because the SMART environment comes with a standard compilation system and some optimizations can only be made at run time. In particular, those run-time optimizations that belong to the Ada tasking which is not supported in the SMART environment are systematically transformed into "null" procedures. For instance the masters identification in certain situations of separate compilation is detected at run time, the body of corresponding ARTS subprograms is then provided as "null" procedures. These "null procedures " must be seen as bad optimized code rather than dead code since they are always covered by the execution of the program. In the current implementation a "null procedure" consists of 14 bytes (for Motorola MC 680x0 processor) performing : a stack push operation, a stack pop operation, and an assembly return instruction. A maximum of six such routines can be embedded in an executable program. However if the separate compilation is not used or if certain rules are followed such as not declaring separate units within a package body, then the executable program will not include any "null procedure" code thanks to the useless subprogram code elimination.

3. subprograms raising an exception : Certain key words are prohibited, for instance "delay", "accept", ... When compiling in SMART environment, warning messages are issued by the compiler if such key words or not supported Ada features are used in the source of the program. The user must then modify his program before linking it. However if he goes to the execution without modification, an exception will be raised for any of not-supported feature which is executed. It is important to note that if the program is compiled with no warning message then the ARTS subprogram will not include those ARTS subprograms raising an exception, thanks to the useless subprogram code elimination provided by the standard compilation system. This ensures that no dead code is embedded.

4. Subprograms to perform elementary arithmetic and logic operations (division, exponentiation, modulo...): These subprograms are not embedded if they are not called (because of the useless subprogram code elimination). If they are needed by the program these predefined operations can and must be preferably re-defined by the user in order not to import the predefined run-time subprograms which might not meet the certification procedures standards.

That way, under a certain programming style the final executable program does not contain any instruction belonging to the Ada run-time system apart from the necessary code to start the program (e.g. to perform the calls to the library units elaboration code). This portion of code is executed just once, so the tests required by certification procedures just consist in proofing that this portion of code is covered with no side effect.

To facilitate the functional and the structural coverage tests of the SMART Executive Alsys provides a certification kit including source and design information about SMART code, allowing a "Glass Box" approach adapted to the certification process.

7.2 C-SMART

The Alsys C-SMART system is a unified toolset enforcing appropriate Ada subset rules and incorporating a certifiable run-time system. It comprises two main parts:

- The C-SMART Ada compiler is a normal Alsys compiler (and thus validated) but which includes a user option to reject programs which use features of Ada outside the prescribed deterministic subset. Any construct whose worst case time of execution and space requirement cannot be predicted is thus excluded with a warning message.

- The C-SMART Executive which is a specially adapted version of Alsys normal run-time system. (C-SMART is actually an acronym for Certifiable SMall Ada Run Time from which the system gets its name).

The subset supported by C-SMART Ada is actually somewhat tighter than that outlined above. Thus there is no tasking apart from the delay statement and no user exception handling. The only access types allowed are at the outer level, must have constrained objects and static collections (avoiding deallocation). Also a number of facilities which require the heap for implementation are forbidden. The result is that the run-time system is much simpler than for full Ada.

The associated C-SMART library system ensures that all the units in a program conform to the rules; the Alsys multilibrary features are still available with the additional constraint that all sublibraries must have the C-SMART library as their ultimate parent.

The C-SMART Executive is, naturally, itself written in C-SMART Ada and is certified and is delivered with all the documentation required as a component of a certified system.

The final outcome is that the user writes the safety critical program in C-SMART Ada; the linked program will then include the C-SMART Executive (strictly only those parts required by that program). The documentation required for certification is then comprised of that specific to the application, and developed by the user, plus that part relating to the Executive, and supplied by Alsys.

The Alsys C-SMART approach corresponds to the Ada subset which restrictions are as follows.

- *Operations on Discrete Types*
  The attribute T'IMAGE is not allowed.
  The attribute T'VALUE is not allowed.

- *Array Types*
  Unconstrained array types are not allowed.

- *Index Constraints and Discrete Ranges*
  The declaration of large array objects are allowed only at the global level.

- *The Type String*
  Because the type STRING is a special case of one-dimensional arrays, the restrictions describe above apply as well on STRING.

- *Record Types*
  Large record objects are allowed only at the global level.

- *Discriminants*
  Records with discriminants are said to be large records if for certain values of the discriminants, the record objects become large objects.

- *Discriminant Constraints*
  Large and constrained records are allowed only at the global level.

- *Access Types*
  Access type definitions are allowed only at the global level.
  Access type definitions with a 'STORAGE_SIZE clause of 0 are allowed anywhere.

- *Aggregates*
  Only static aggregates are allowed.

- *Logical Operators and Short-circuit Forms*
  The logical operations (OR, AND, XOR) on unpacked arrays of boolean components are not allowed.
  The logical operations (OR, AND, XOR) are allowed for PACKed arrays of boolean components whose size is known at compiler time and is either 8, 16, or 32 bits.

- *Binary Adding Operators*
  The catenation operation (&) is not allowed.

- *Highest Precedence Operators*
  The logical negation (NOT) on unpacked arrays of boolean components is not allowed.
  The logical negation (NOT) is allowed for PACKed arrays of boolean components whose size is known at compile time and is either 8, 16, or 32 bits.

- *Allocators*
  Only objects of a global collection of fixed size elements, with a 'STORAGE_SIZE clause can be allocated using an allocator.

- *Task Specifications and Task Bodies*
  Task specifications and task bodies are not allowed.

- *Task Types and Task Objects*
  Task types and task objects are not allowed. Therefore the following are not applicable:

  Task Execution - Task Activation, Task Dependence - Termination of Tasks, Entries - Entry Calls and Accept Statements, Select Statements, Task and Entry Attributes, Abort Statement, Example of Tasking, Exceptions Raised During Task Communication, Exceptions and Optimisation.

- *Priorities*
  The pragma PRIORITY is not supported.

- *Shared Variables*
  The pragma SHARED is supported. Its only effect is to disable tracking optimisation on the variables being shared.

- *Exception Declarations*
  Exception handlers are not allowed. As a consequence any exception being raised is fatal. It is the user's responsibility to provide post-mortem procedures to catch any possible exception. Also *Exceptions and Optimisation* are not applicable.

- *Raise Statements*
  Raise statements are not allowed.

- *Unchecked Storage Deallocation*
  The restrictions on unchecked storage deallocation are those resulting of the Allocation.

- *Input-Output*
  No Input-Output as defined in the Reference Manual is supported by C-SMART.

- *Predefined Language Attributes*
  The following attributes are not allowed.

  P'COUNT, P'IMAGE, T'VALUE, P'TERMINATED.

8 Certified Applications

8.1 BSCU (Braking and Steering Control Unit) for the landing gear system of A330-340 Airbus

This software has been developed within Thomson CSF / D.O.I. for Messier Bugatti, responsible for the landing gear for British Aerospace. It has been developed in less than 30 months, and has 320 000 lines of code, out of which 140 000 written in Ada.

The BSCU calculator ensures the braking and anti skid functions of the eight wheels and the orientation of the front wheel-axle unit of all A330-340 Airbus. As for the software, this calculator has been entirely designed by Thomson CSF / D.O.I. It comprises a redundant architecture with 10 Motorola microprocessors (68000, 68020, 68332) divided in 10 numeric and analogic boards.

The presentation was an important step towards the certification of the landing gear of A330-340 Airbus. The development methods according to the international standard (DO 178A) guarantees the reliability and safety of the system.

This software has been successfully presented to the European Certification Authorities JAA (Joint Aviation Authorities) in September 1992.

8.2 The FCDC Computer

The FCDC Calculator (Flight Control Data Concentrator) is one of the calculators used for the electric flight commands of A330-340 Airbus.

Its functions are as follows:

to concentrate various information coming from the other electric flight commands calculators and forward them to the information display systems in the cockpit, to manage the alarms and "plane" maintenance.

to analyse the behaviour of the other electric flight commands calculators, diagnosis and locate possible breakdown and store them in order to help the ground maintenance teams.

to manage the dialogue with the maintenance teams for the analysis of events occurred during the flight and send-up specific tests to the electric flight commands calculators.

The criticality of some of the above functions has led the Certification Authorities to classify the software of the FCDC calculator at level 2A (the reference documents for all software certification aspects have defined 4 criticality levels :1, 2A, 2B and 3 - level 1 is for software that has to meet a high level of requirements).

The FCDC calculator is based on a Motorola 68000 microprocessor.

The FCDC Software is developed by the Technical Division of Aerospatiale, the FCDC software has a size of 330 Kbyte for 73,000 lines of source code.

It is composed of two parts:

- The first part (65,000 lines) corresponds to the functions of data concentration and breakdown analysis. It has been given a graphic look with a tool named SAO set-up by Aerospatiale for the design of the embedded systems. The corresponding software is automatically generated by a specialised software developed for this purpose in the Ada language.

  The macro-assembler (compatible with the Alsys Ada compiler) has been used.

- The second part (8,000 lines) has been designed with the HOOD method and written in the Ada language.

It corresponds to the management of breakdown detected (filing, storing, ...) and to the dialogue with the maintenance teams.

The Alsys Ada compiler as well as SMART have been used.

The software and the automatic generation tools have received the agreement of the Certification Authorities in December 1992.

8.3 Hydro-Aire for Boeing 777 Brake Control System

Hydro-Aire has selected Ada software development tools for the development of the brake control system for the Boeing 777 aircraft. Hydro-Aire will be using Alsys AdaWorld cross compilers with the Alsys SMART Executive and Certification package. The certification package will be used by Boeing during FAA certification of the brake control system due to the use of commercial off the shelf (COTS) software, specifically, the Alsys run-time executive.

Hydro-Aire is using Alsys AdaWorld Ada compilers on Hewlett-Packard HP9000/300 platforms, targeting the new Motorola 68333 micro controller. Each 777 aircraft's brake control system will include ten micro controllers of which two are Motorola 68333 micro controllers, programmed primarily in Ada. The Motorola 68333 micro controllers will control the built-in-test (BITE) and Auto brake functions. The BITE includes both an on-line interface to the central maintenance computer and an off-line maintenance capability. The Auto brake automatically applies the correct amount of brake pressure during landing, as well applying the maximum amount of braking during refused takeoffs (RTOs). The brake control system also includes additional hardware and software to provided anti skid capabilities.

9 Conclusion

Alsys is currently and will be even more in the future committed to provide solutions for applications which need to be certified.

The current both marketing and technical researches are performed following two main directions:

- take benefit and experience of the two approaches SMART on 68K and C-SMART on Intel to provide a unique solution which offers the best of the two approaches
- investigate the possibility to extend the Ada subset so as to be used in certified applications; the corresponding runtime will have to be defined concurrently.

The SMART/C-SMART solution will of course allow the two existing methods used for certification:

- the "glass box" approach, based on the reduction of the Ada subset defined by SMART,
- the "black box" approach, based on the availability of the complete package (i.e. code + documentation required by the DO178B standard)

The advantage of the future unique solution will be its flexibility. Thanks to precise mapping between the features in the Ada subset and the corresponding pieces of code in SMART, a customer will have the possibility to use the "glass box" approach with a rather large Ada subset (but limited to SMART subset anyway) or the "black box" approach for a reduced Ada subset.

The more ambitious solution, that is the definition of an Ada subset larger than the SMART Ada subset, should bring an answer to the problem of the increasing complexity of the applications to be certified and the even bigger complexity of the certification process.

Some Ada features might not be recommended for safety critical applications, because the necessary predictability is not guaranteed by the language but relies only on the implementation.

Current studies will show which are the "safe" features. the features only "safe" if the implementation allows it and the "unsafe" features which are to be avoided for a safety critical application.

The purpose of the current study is really to identify the second category (i.e. the features "safe" because the implementation is) because it really depends on the know-how and experience of a major Ada vendor such as Alsys. This is the main advantage since it is complementary to an only theoretical approach, as some were already used in the past.

References

[1] "Software Considerations in Airborne Systems and Equipment Certification", RTCA DO-178A / EUROCAE ED-12A, October 1985.

[2] I.C. Pyle, "Developing Safety Systems: A Guide Using Ada", Prentice Hall 1991.

Discussion

Question W. MALA

During the presentation, the author stated that the real-time executive could be "certified" for use in safety critical applications. I believe that this position is misleading and does not reflect the overall problem of software certification in safety critical applications. A Runtime Executive on its own can never be "safety critical" and can therefore not be certified.

What has been called "certification of Ada Real-Time executive" is no more than a validation of the Real-Time executive against a defined functionality. The certification must include the functionality and behaviour of all components comprising the "safety critical software function", consisting of application software, Ada compiler and real time executive. It would not make sense, if the real time executive would have been certified and the Ada compiler not.

Reply

The author accepted the comment.

Design of a Thinwire Real-Time Multiprocessor Operating System

Charles Gauthier
Software Engineering Laboratory
Institute for Information Technology
National Research Council of Canada
Ottawa, Canada, K1A 0R6

1. SUMMARY

As more and more capabilities are added to avionics and other real-time or embedded systems, it becomes necessary to increase the processing power of the underlying executors. At any point, technology imposes limits on the available performance of individual processors. Increasing the computing power beyond those limits requires the use of multiple processors.

However, developing a multiprocessor real-time system is often difficult and expensive. The lack of sophisticated software tools makes the development process extremely tedious and error prone. In addition, many architectural difficulties must be overcome. Some real-time systems must be implemented on top of existing machines that do not lend themselves well to multitasking systems that depend on shared-memory. Other real-time systems are built from components that present particular architectural problems. Data caches, in particular, introduce the *cache consistency* problem. Consistency protocols designed to keep the caches consistent are not always usable, and they often introduce substantial performance penalties. Without consistency protocols, because of *false sharing*, shared variables may become inconsistent even if mutual exclusion mechanisms are used.

This paper presents an implementation of a multiprocessor multitasking real-time operating system on difficult architectures. The development of applications on this operating system does not require any special software development tools such as special compilers. Furthermore, the applications can be ported to a very wide range of multiprocessor architectures. The principles of operation of the operating system can be applied to the implementation of an Ada runtime environment, if some restrictions are observed.

2. INTRODUCTION

Developing a multiprocessor real-time system is generally more difficult and more expensive than developing a uniprocessor real-time system because of the added synchronization and communication problems. The lack of specialized software tools makes the development process extremely tedious and error prone. Some programming languages do provide multiprocessing abstractions, but they are rarely used in the development of industrial and military systems. Most languages used to implement multiprocessor systems do not support the notion of multiple tasks, much less multiple processors. Even Ada, which provides a tasking abstraction, does not support multiprocessing at the language level.

In addition, most industrial and military real-time systems are built from existing architectural components. These components range in complexity from single microprocessors to complete multiprocessors, and include board-level components. Many of the existing components present serious obstacles to their use in multiprocessors. For example, most high-performance 32-bit and 64-bit microprocessors feature on-chip or close-coupled off-chip instruction and data caches. The use of data caches in shared-memory or tightly coupled multiprocessors introduces the cache coherency or cache consistency problem, which is the problem of insuring that all processors operate upon the most up-to-date values of variables. While many processors provide hardware support for cache consistency, their use with existing buses, such as the VMEbus, often means that cache consistency protocols cannot be used. Without cache consistency protocols, false sharing can occur. In false sharing, variables that are not shared at the programming level become shared at the hardware level. False sharing means that traditional mutual exclusion mechanisms no longer control access to shared data. As another example, some existing multiprocessors offer a mix of shared and private memory, complicating the implementation of shared-memory systems because data to memory allocation must now be controlled.

The adoption of the thinwire model solves these difficulties. The thinwire model is described in detail in the next section. Section 4 discusses in detail the reasons one might adopt the thinwire model. Section 5 then describes an actual implementation of the thinwire model in a shared-memory multiprocessor. This implementation avoids the cache consistency problem.

3. DEFINITION OF THINWIRE

The term *thinwire* multiprocessor is defined in this paper as a message-passing abstract machine rather than as a physical machine. Because it is an abstract machine, the underlying physical machine may be a shared-memory or tightly coupled

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

multiprocessor. The characteristics of the thinwire abstract machine are:

- The thinwire multiprocessor is a fully connected machine in the sense that a processor may send a message to any other processor. A connection need not be direct; intermediate nodes may exist in a path between two processors. However, routing of messages through intermediate nodes should be performed without the intervention of the thinwire machine; it could be done by hardware or by layers of software below the thinwire abstraction.
- The links in a thinwire multiprocessor are assumed to be reliable in the same sense as a backplane bus is assumed reliable, i.e. messages are never lost and are always delivered in the same order in which they were transmitted. This does not mean that errors of transmission never occur, but any error in transmission is immediately signaled by hardware. Repeated errors in transmission indicate a system-wide fault that either causes a system shutdown or requires the use of fault-tolerant techniques to mask the fault, such as a switchover to a backup communication link. This assumption permits the use of very lightweight protocols for reliable transmission. There is no need for heavyweight protocols designed to deal with lost messages or the out-of-order delivery of messages.
- Processor failures are considered system-wide faults that bring the entire system down unless fault-tolerant techniques are used to mask the faults or recover from them.
- Processors are not autonomous, i.e. the nodes are not powered up and down independently nor are they reset independently.
- The thinwire multiprocessor runs a single multiprocessor operating system on all processors. This does not mean that all processors share a single copy of the kernel code or data structures. Given the class of machines the thinwire abstract machine is targeted to, it is expected that, in most cases, each processor would run its own copy of the kernel code and would maintain its own copy of the global data structures. These multiple copies of global data structures are kept synchronized by mechanisms implemented in the lowest level of the kernel—on top of the message-passing mechanisms in true message-passing machines, and in parallel or below the message-passing mechanisms in shared-memory machines.

A thinwire system is not a multicomputer or a distributed system. Both multicomputers and distributed systems are computer systems in which nodes[1] run a standalone kernel and multiple independent application programs.

4. WHY BUILD THINWIRE SYSTEMS

As stated, the thinwire approach gets around some architectural problems without resorting to specialized software tools. Some of the architectural difficulties addressed are:

- The multiprocessor is not a shared-memory machine.
- The multiprocessor is built from heterogeneous processors.
- The multiprocessor is a uniform shared-memory machine, but caching problems arise when memory is shared arbitrarily.
- The multiprocessor is a uniform shared-memory machine, but performance degradation occurs because cache consistency protocols are used.

These difficulties are discussed in detail in the next sections.

4.1 The multiprocessor is not a shared-memory machine.

The most obvious reason, and the most trivial, is that the underlying machine does not have shared memory. This class of multiprocessors includes true message-passing multiprocessors like the Transputers [1][2] and the Message-Driven Processor [3]. It also includes shared-memory machines in which only a portion of local data memory is sharable or where complete connectivity does not exist between the processors.

It is not rare to find multiprocessors in which only a portion of the address space is shared. Such machines are often built to get around a lack of sufficient address bits. For example, with an Intel 8086 processor, only 20 address bits are available, for a total space of 1 megabyte of directly addressable memory. In a multiprocessor system, implementing a single, global 1 megabyte address space might leave insufficient memory for each processor to store its programs and data. It is preferable to give each processor its own private space and to provide a subset of the 1 megabyte space as a global memory space. In such an architecture, it might be preferable to treat the multiprocessor as a message-passing machine to present the application layers with a uniform programming paradigm. If the shared memory is made visible to the program, programmers or compilers have to track the sharable data and allocate it to the sharable regions of memory.

Many single board computers have only private memory as a cost reduction feature. These boards have been designed specifically for applications that require a single general purpose processor and rely on other cards to provide I/O devices or extra memory. These cards cannot be combined to form a multiprocessor unless global memory cards

[1] A node in such a system may be a uniprocessor or a multiprocessor.

are used. In systems built from such cards, only a subset of the memory is shared.

Other thinwire candidate machines are the shared-memory multiprocessors in which the shared modules are not accessible to all processors. One such machine is the TELDIX multiprocessor developed by TELDIX GmbH and used on the PANAVIA Tornado aircraft [4]. This architecture is illustrated in Figure 1. Currently, applications for this multiprocessor are written in SD-Scicon Ada. When building a multiprocessor Ada application, implementors must describe the underlying architecture of the executor to an automated application builder and must specify the desired processor access to program units. An allocation phase during compilation and linking attempts to place these program units in memory locations accessible by the desired processors. If this is not possible, executable images are not created. The failure to transform the source programs into executable images is entirely attributable to the support for shared memory; in a thinwire machine, messages could be forwarded automatically to the destination processor.

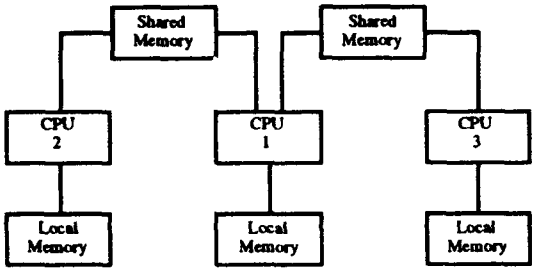


Figure 1: TELDIX computer organization

4.2 The multiprocessor is not a homogeneous machine.

It is not rare in embedded systems to mix processor types in a single multiprocessor. Indeed, specialized processors such as Digital Signal Processors (DSPs) may provide specialized services. For example, in some radar and sonar systems, DSPs filter digitized echo data in real-time before passing the information on to more conventional processors. Some DSP chips, such as the Motorola DSP56000 and the members of the Texas Instruments TMS32000 family, were designed to communicate with other processors using a message-passing scheme [5][6][7]. One approach to communicate with these devices is to treat them as I/O devices. This may make sense with the earlier devices such as the TMS32010, which has a maximum of 8k bytes of program memory and 288 bytes of data memory. However the newer DSPs, such as the TMS32040, have more computing power and, what is more important, large address spaces. The TMS32040 has a full 32-bit data and instruction address space and 6 communication links to realize a variety of interconnection topologies [8][9]. This means that it is now feasible to run multitasking multiprocessor operating systems on these devices. In many situations, it may not make sense to do otherwise; the new high-power DSPs are relatively expensive, so they must be used to their full potential. One way to do this is to keep the processor busy doing multiple signal processing functions or processing multiple data streams in round-robin fashion. It is not inconceivable even to use these processors in priority driven systems. From a system design perspective, it is desirable to have a uniform operating system running on the overall system to keep the view of the system simple and consistent. Given that some manufacturers do not provide shared-memory facilities to exchange data between DSPs and between DSPs and other processors, a thinwire implementation is very useful. Other considerations may also lead to the adoption of a thinwire implementation, such as different data representations on the different types of processors. If the type of the objects being exchanged in messages was somehow known to the kernel, data representation conversions could take place transparently and correctly, which might not be the case if the process was left to programmers.

The previous discussion applies to a variety of specialized processors used in embedded real-time systems, such as graphics processors.

4.3 Caching problems.

Caches have been used successfully in uniprocessors to reduce the average access time to memory by keeping copies of data and instructions in faster memory. In multiprocessors, caches can also be used to reduce the need for a processor to fetch data from memory by keeping copies of data and instructions in the cache close to the processor. This not only decreases the average memory access latency, but also the average memory and communication contention [10].

Before presenting problems associated with caching, and particularly the false sharing problem, which as motivated the work reported in this paper, a brief review of cache organizations and cache coherency may be of benefit to the reader. An excellent survey on caching can be found in [11].

A caching system breaks main memory into a number of usually equal size chunks called *blocks*. These blocks are then copied individually into the cache, into cache *lines*. Any reference to a location in a block causes the entire block to be copied into a cache line, provided that the block was declared cacheable. A *tag*, the address of a block, is stored in a line with every block to distinguish which block is in a given line. Whenever a memory access is made, the cache must compare the block address to the tags in the cache. If a matching tag is found and the block is valid, a *hit* has occurred, otherwise the data is not in the cache and a *miss* has occurred.

During a read from memory, if a hit occurs, the data is supplied by the cache and the bus cycle to memory can be preempted, unless the cache consistency protocol requires a memory cycle. During a write, several possibilities can occur regarding the updating of the caches and memory.

If a hit occurs, the data can be written only to the cache until the data is explicitly flushed to memory or until the cache line is reused for another block. This mode of operation is called *copyback* or *writeback*. An alternative is to write the data both to the cache and to memory. This is called *writethrough* or *storethrough*. If a miss occurs, the cache content may be updated in copyback or writethrough mode, or the data may be written to memory only.

The use of data caches in multiprocessors introduces the *cache consistency* or *cache coherency* problem, which is the problem of keeping all the cached copies the same. To illustrate the problem, consider a two-processor shared-memory multiprocessor. Assume processor A reads a variable. The value of that variable is copied in the cache of processor A. Now processor B reads the same variable and copies it into its cache. If processor A now writes to the variable, processor B's cached copy of that variable becomes obsolete. Such an out-of-date value is called a *stale* value.

Some caches can monitor bus traffic and take steps to maintain consistency. This monitoring is called *snooping*, and the set of rules followed to maintain cache content consistent is called a *protocol*. For example, an *invalidation-based* protocol invalidates a cache line when a cache detects a write to a memory block that it has currently cached. If the block is referenced again, the up-to-date value is fetched from memory. An *update-based* protocol updates a line rather than invalidate it as a result of snooping. Some protocols require that caches operate in writethrough mode so all write cycles are snooped by all caches. Other protocols require that caches snoop read cycles to determine if other caches have copies of a block; if so, to allow other caches to invalidate or update a block, a copyback cycle must occur every time a block held in two or more caches is modified.

The cache coherency schemes described so far depend on the existence of a global bus to snoop bus transactions. Because of the existence of buses in all multiprocessors, these schemes are the most common and possibly the only ones used in embedded systems. *Directory-based* caches also exist. With these caches, the tags of the various caches are kept in global directories. This means that a cache can determine if an invalidation or an update is necessary; a snoop cycle is not required, reducing contention for the bus. Directory-based caches are not easily implemented in commercial bus-based systems and will not be discussed further.

Having completed a brief review of caching, problems it introduces can now be examined.

One problem is that most commercial boards have no bus snooping capability at all. For example, when a processor accesses its local memory, a bus cycle is not generated on the backplane bus (VMEbus, NuBus, etc.). Hardware cache consistency is impossible in such systems [12][13]. For performance reasons, it is preferable to generate snoop cycles only for shared data rather than all data. Few buses carry the necessary signals to distinguish the different types of accesses. The FutureBus [14] and newer FutureBus+ [15][16] are two exceptions [17][18], but they are still relatively unused. The current bus systems (VMEbus, NuBus, Multibus I and II), which do not support cache consistency well, are not likely to be abandoned soon. Furthermore, there are many microprocessors that do not provide hardware-coherent caches. The Motorola 68030 and Intel 80486 have no built-in hardware cache consistency capabilities at all[2] [19][20], whereas the 68040, which supports both writethrough and copyback, must be used in writethrough mode in multi-68040 systems if hardware cache consistency is used [21].

The solutions adopted by most implementors to solve the cache consistency problem is either to avoid it altogether by not caching shared data or by insuring that cached copies of shared data do not exist in more than one cache at any given time. This can be done by serializing access to the shared data using any of the available mutual exclusion mechanisms and by flushing and invalidating the cached global data before releasing the mutually exclusive lock on it.

The first solution—not caching the shared data—has been used in the IBM RP3 [22]. However, it has two problems. The first is that it can lead to gross inefficiencies because every access to shared data incurs the full latency to memory, not to mention that it increases contention for memory. A study by Owicki and Agarwal confirmed that the performance of this *no-cache* solution is worse than any other scheme is most cases and is abysmal if there are a large number of references to shared data (17% of the instructions in the study)[23][24]. The second problem is that shared read-write data must be identified and made non-cacheable. This process should be done by the compiler in cooperation with the linker and runtime library, but most commercial real-time system development is done with existing languages and development systems, most of which, if not all, do not perform this function. In many cases, each processor image is compiled and linked independently, so that automatic program analysis to detect the sharing of data is impossible. In almost all cases, programmers are responsible for identifying all shared data and placing it in the appropriate locations, an exercise that is error prone.

The second solution—insuring that cached copies of shared data do not exist in more than one cache at any given time—is preferable from a

[2] In fact, the Intel 80486 does provide inputs into the cache so lines can be invalidated selectively by external hardware. Intel supplies an external cache controller with snooping capability as a single VLSI component, the i82495. The Motorola 68030 does not provide external cache control, so that hardware cache consistency cannot be implemented.

performance point of view. Furthermore, since the data is shared, it is likely to be already protected by some form of mutual exclusion, so nothing special need be done. Unfortunately, that solution ignores the *false sharing* problem [25]. There is very little literature on it, although it has been known to exist for years.

All microprocessor caches copy blocks of contiguous memory locations into the cache lines rather than individual objects. The problem is that cache lines are totally unrelated to the software objects in memory. This is illustrated in Figure 2. That figure shows a portion of memory corresponding to 32 bytes starting on a 16-byte boundary and assumes a 16-byte cache block size. If any one byte within one of these blocks is referenced as part of a cacheable memory cycle, all bytes in that block are pulled into a cache line. If the reference is to a multi-byte entity that straddles two blocks (shown as the shaded area in figure 2), both blocks are copied into the cache. For example, a reference to any number of bytes in object 1 causes the topmost block to be cached, including the first bytes of object 2. If a reference is made to the bytes in object 2 shown as the small shaded rectangle, both the first and second blocks will be copied into the cache.

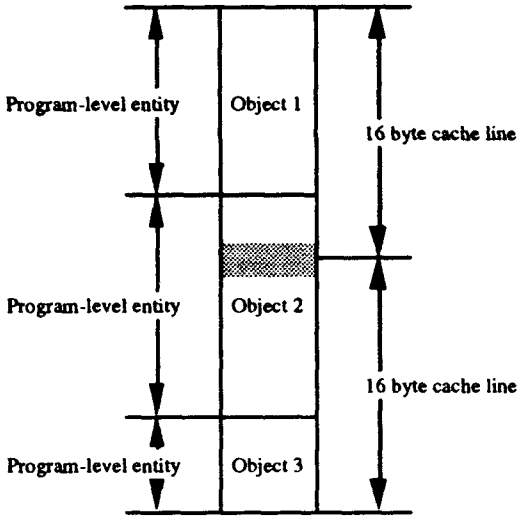


Figure 2: Relationship between program objects and cache lines.

This scheme leads to the sharing of objects at the hardware level, something that is not apparent at the software level. To see the problem, assume that object 1 and object 2 are both protected by some mutual exclusion mechanism and that the mechanism is properly used. Now assume that the fragments of pseudo-code shown in Figure 3 are executed concurrently.

The illustrated program is logically correct. Each thread first gains exclusive access to the object before modifying its value. The locking mechanism can be any of the traditional mutual exclusion mechanisms: spin locks, suspend locks, semaphores, etc. The object is then flushed from the cache so it can be accessed in memory by other processors, and the cached copy is invalidated before the lock is released. This insures that either processor will get the object from memory and not from its cache the next time it acquires the lock, in case some other processor has modified the value in the interval. Unfortunately, when processor 2 caches object 2, it is also caching object 1. Similarly, when processor 1 caches object 1, it is caching a portion of object 2. One of the two objects will be corrupted in memory when the flushes occur; object 1 is corrupted when processor 2 flushes after processor 1, *else* object 2 is corrupted when processor 1 flushes after processor 2.

processor 1	*processor 2*
lock(object1);	lock(object2);
read(object1);	read(object2);
modify(object1);	modify(object2);
write(object1);	write(object2);
flush(object1);	flush(object2);
unlock(object1);	unlock(object2);

Figure 3: Two logically correct threads that illustrate the false sharing problem.

There is no way to determine that false sharing is occurring short of analyzing the memory layout of program data. As with the first solution, compilers and linkers that control data layout at this level are not readily available, so that the process would have to be performed manually. This is a highly tedious and error prone process with any non-trivial application program. If the programmers have knowledge of the data layout of the objects and have noticed that there is a problem, they could lock both object 1 and object 2 in a single lock operation. This approach is not a serious option, because the layout might change every time the program is modified. To keep the program as efficient as possible, the locking code has to be changed every time there is a code change. Otherwise, to avoid modifying the locking code, all objects susceptible to being falsely shared together have to be locked together, thus seriously reducing performance by serializing access to data structures that otherwise could be accessed concurrently. In some machines, special access instructions that do not cache the accessed block are available. However, they do not solve the problem; because of false sharing, an access to a cacheable item might bring into memory an item that is not supposed to be cached. This item would be returned to memory whenever its containing block was flushed, obliterating the value in memory, which might have been changed.

In a thinwire system built on top of a shared-memory multiprocessor, it is possible to control the size and alignment of storage blocks such that portions of two different blocks do not lie in the same cache line. As long as dynamically created objects are allocated in such blocks, they can be

shared safely among different processors. It is also possible to control the layout of static kernel data structures, so that a thinwire implementation on such a machine could take advantage of the shared memory to achieve a high degree of performance. What cannot be shared are variables declared in the program text, e.g. global variables and local variables identified by pointers, because of the inability to control the storage of such objects.

4.4 Performance reasons.
What is not generally recognized, even though the topic is covered in standard textbooks [26], is that the sharing of read-write data can also lead to degradation of performance in cache coherent systems. There are four potential sources of performance degradation related to the use of coherent local caches:

1. Degradation of the average hit ratio due to block invalidations. In systems that use an invalidation-based coherency protocol, blocks will have to be fetched from memory repeatedly following invalidations. Some of these protocols also require the use of the writethrough memory update policy, a further source of performance degradation.
2. Multiple copyback cycles due to block modifications. In systems that use a copyback-based coherency protocol, every time a processor modifies a block cached in other processors, a copyback cycle is generated to invalidate or update the other copies. However, a copyback operation with update in the other caches is more efficient than a copyback operation with invalidate because the modified block is pulled into the other caches during the copyback.
3. Traffic between the caches to detect inconsistencies. In systems with snoopy caches, contention for the bus can become very significant because all transactions to cacheable shared locations must be broadcast to the other caches. Thus, the reduction in bus contention obtained by keeping copies of data close to the processors is not achieved, even for private data. This has to do with the fact that snooping control is generally established at the page level, as is caching control, and with the fact that private data is often mixed with the shared data, so that snooping cycles are generated even when not needed.
4. Contention for access to directories. The directories are shared global resources, so contention is unavoidable, although some clever directory designs may reduce it over a single ported non-interleaved directory. Such a design would also imply a more sophisticated interconnection network than a shared bus, increasing the cost of the system.

The first two sources of performance degradation cannot be avoided, even if access to and caching of mutable shared data is properly managed, because of false sharing, i.e. invalidations or updates can result from modifications to distinct data elements that lie in the same memory block. At best, performance degradation can be reduced by managing the true shared data. This performance degradation due to sharing has been studied by Weber and Gupta [27] and by Agarwal and Gupta [28].

The last two sources of performance degradation are built into the coherency solutions and cannot be eliminated. Worse, the performance degradation scales with the size of the multiprocessor as the snoop traffic increases, or as the traffic and contention for the directory increases. At some point, hardware cache consistency destroys any benefit of increasing the number of processors. It should be possible to increase performance by avoiding the cache consistency problem altogether, thus eliminating the need for coherency protocols. This is certainly true as the number of processors increases, especially with snoopy protocols over a single shared bus.

5. AN IMPLEMENTATION

The last section argued that not all multiprocessors are shared-memory machines or homogeneous machines. The last section also argued that, even in a homogeneous shared-memory multiprocessor, cache consistency protocols cannot always be used and that, even if they were used, their cost is often unacceptable. That section also showed that, without cache consistency protocols, false sharing makes the usual mutual exclusion mechanisms unusable unless the layout of shared data is carefully controlled. This control must be done at the application level, because existing compilers do not do it. It was shown that these difficulties could be overcome by treating the multiprocessor as a thinwire machine.

This section looks at the implementation of a thinwire abstract machine for a shared-memory homogeneous multiprocessor. The target system is a MC68040-based multiprocessor built from dual-processor Synergy SV420 VMEbus cards. While the target system does have shared-memory, cache consistency protocols cannot be used to keep all caches consistent; cache consistency protocols can synchronize only the data caches of the two processors on a single card. The thinwire approach was selected in this case to avoid the cache consistency problem. Application programs are prohibited from using shared-memory for data exchange; all data are shared through message-passing. Memory is shared below the application level, i.e. in the kernel. This sacrifices portability of the kernel to non-shared-memory multiprocessors, but it does achieve higher performance. The solution should apply to any similar type of multiprocessor without modifications.

The implementation is integrated into the latest release of the Harmony operating system developed at the National Research Council of Canada. Harmony is a real-time multitasking multiprocessing operating system [29][30].

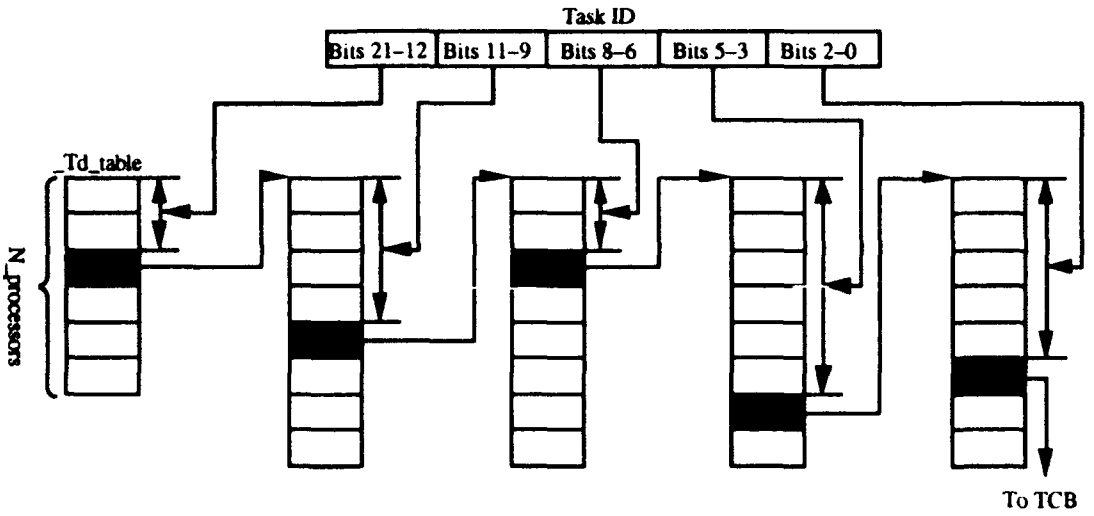


Figure 4: _TD_Table data structure.

Currently, versions of Harmony exist for the Motorola 68000 family of processors, the Motorola 88000 family, and the Intel 8086 family. In the past, Harmony was successfully ported to the National Semiconductor NS32000 and the Digital Equipment Corporation VAX. Current systems work with the VMEbus, Multibus, and NuBus.

Harmony uses a microkernel and several system servers to provide services to lightweight application tasks. Harmony itself is written mostly in C, with some machine-specific portions written in assembler. Applications running on top of Harmony may be written in almost any language. Because each processor image is compiled and linked independently, special software development tools are not required.

To understand the implementation, is it necessary to understand the Harmony message-passing primitives. Harmony uses the send-receive-reply mechanism for communication and synchronization[31][32]. In this mechanism, a task that sends a message to another task blocks until the task sent to replies. A task that receives a message blocks until a message is received[3]. A receiving task can receive from any task or from a specific task. The exact operation performed depends on the value of a parameter in the receive function call. The receiving task can return data to the sending task in a reply message. The reply call is not blocking. The reply is required to unblock the sending task. A receiving task can unblock a sending task without sending any information simply by issuing an empty reply message.

These message-passing primitives can be compared to the Ada rendezvous mechanism [32]. A send is analogous to an entry call. A receive is analogous to an accept of an entry. Both the Ada entry call and the accept are blocking, as are the Harmony send and the receive. The difference is in the reply. In Ada, the reply is implicit and automatic when the accepting task exits the scope of the accept statement. In Harmony, the reply is an explicit call which can occur at any point after a message was received from the task being replied to. The possibility of issuing out-of-order replies makes the Harmony message-passing primitives more flexible than Ada's.

The thinwire implementation of Harmony must control the content of the caches when dealing with shared data. Fortunately, the only data that is shared between processors are the messages, the task control blocks, and a few kernel data structures. All dynamic data structures are allocated in memory blocks that are an integral number of cache lines in size and aligned on cache line boundaries, thus preventing unrelated objects from lying in the same cache lines, i.e. avoiding false sharing problems. Consequently, it is sufficient to enforce some form of mutual exclusion on the access to the shared structures, and to flush and invalidate the cache lines at the proper points to maintain consistency.

Because each processor image is built independently, the shared kernel data structures are built at system startup time from information in each image. These structures are built in dynamically allocated memory blocks, so that false sharing does not occur. All but one of the shared kernel data structures are read-only. Mutual exclusion is not required for read-only shared data. By building the read-only data structures before enabling caching, cache consistency problems do not arise.

Only one data structure, the _TD_table illustrated in Figure 4, is variable. This data structure maps task identifiers (32-bit integers assigned to tasks when they are created) to pointers to the corresponding task control blocks (TCBs). As

[3] Harmony also provides a non-blocking receive. It returns immediately if a message cannot be received.

shown, this data structure is implemented as a tree. The first level of the tree is an array with as many elements as there are processors in the system. This array is created at system startup time, before caching is enabled, and is then read-only, so that mutual exclusion is not required, and caching this level of the tree in different processors is not a problem. Each element of this array is a pointer to a dynamic subtree. One subtree is allocated per processor. In the particular implementation of the _TD_table illustrated, each level of the subtree is an array of 8 elements, each one indexed by a 3-bit wide field in the task identifier (task ID). The leaf nodes of each subtree contain either a pointer to the TCB corresponding to the task identifier, or an indication that the task identifier does not correspond to an existing task. Only the processor on which the subtree is allocated updates the subtree. The update procedures update the subtree branches atomically and flush any changes from the caches, thus insuring that the up-to-date data is in memory. The procedures that read the elements of the subtree invalidate the corresponding cache lines to ensure that up-to-date values are always read. Again, because of the atomicity of the updates and the single writter, mutual exclusion is not required. The flushes and invalidates, in combination with control of the data layout, are sufficient to maintain a consistent view of this structure.

The task control blocks are dynamically allocated. Like all dynamically allocated data, the TCBs are allocated in blocks that are an integral number of cache lines in size and aligned on cache line boundaries, thus insuring that false sharing does not occur. Any processor can read and write a TCB. An ownership protocol in used to implement mutual exclusion. For example, when a task on one processor sends a message to another task on a different processor, the ownership of the TCB of the sending task is transferred to the processor of the receiving task so that processor can change the state of the sending task. The processor that has ownership of a TCB is the only processor that can modify it. This ownership protocol is implemented in a state machine at the core of the kernel. Cache flush and invalidate statements are included in the state machine implementation to maintain caches consistent. For example, whenever a processor reads one or more fields in a TCB it does not own, it must invalidate the corresponding cache lines because the fields may change. Without the invalidates, the reading processor may read stale data.

The only application level shared data are the messages. Messages are allocated either dynamically (in the heap) or as local variables on the stacks of tasks. Messages allocated in the heap are an integral number of cache lines in size and aligned on cache line boundaries. Consequently, false sharing cannot occur, and the blocking nature of the message-passing primitives provides the required mutual exclusion. When sending a message, the message is flushed from the cache. When receiving a message, the storage allocated to hold the received message is first invalidated to insure that the actual sent message will be read from memory. These steps are sufficient to ensure that dynamically allocated messages are always consistent. Messages allocated on the stack are also dealt with properly, although understanding how this is done is more difficult. Indeed, because these messages are allocated by the compiler in the stack frames, they are not an integral number of cache lines in size and they are not aligned on cache line boundaries, so that false sharing can occur. Fortunately, Harmony copies messages from the storage of one task to the storage of the correspondent task. Even in systems with caches, only the actual bytes that make up the message are copied, although complete blocks are pulled into the caches. This copying, coupled with the blocking nature of the message-passing primitives, ensures that any falsely shared data will not be modified. Of course, all this works only if the falsely shared data is not modified concurrently. Harmony guarantees that there are enough bytes placed on top of a stack when a task is created, and enough storage space left on the stack for exception processing, that any falsely shared data must belong to a blocked task. Consequently, concurrent modification of the falsely shared data cannot occur.

The mechanisms described are sufficient to maintain a consistent view of memory on a shared-memory multiprocessor without recourse to cache consistency protocols. There are no mechanisms to prevent application tasks from using shared-memory for communication. However, the kernel cannot support such usage. For example, pointers can be passed in messages, but any access through such pointers is invisible to the kernel, so cache consistency operations cannot be performed automatically. Because shared-memory is not supported at the application level, the multiprocessor is described as a thinwire multiprocessor.

6. CONCLUSION

The paper described several problems that can occur in embedded and real-time multiprocessor systems. The paper showed that these problems can be overcome by implementing the system as a message-passing system rather than as a shared-memory system. The high-performance type of message-passing system proposed is described as a thinwire multiprocessor. This multiprocessor is a virtual machine that can be implemented on shared-memory machines with heterogeneous processors or without consistent caches, and on non-shared-memory machines and true message-passing hardware.

An implementation of the thinwire multiprocessor for a shared-memory machine without consistent caches was described. The principles behind the implementation can be applied to Ada with little if any modifications.

An efficient implementation for true message-passing machines is currently being investigated.

7. REFERENCES

1. INMOS Limited. Transputer Reference Manual, Prentice-Hall International (UK) Ltd (1988).
2. *The T9000 Transputer Products Overview Manual*, INMOS Ltd, SGS-Thomson Microelectronics, First, 1991.
3. Dally, W.J., Fiske, J.A.S., Keen, J.S., Lethin, R.A., Noakes, M.D., Nuth, P.R., Davison, R.E., and Fyler, G.A. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro 12*, 2 (April 1992), pp. 23–39.
4. Collingbourne, L., Cholerton, A., and Bolderston, T. *Distributed Ada: Developments and Experiences. Proceedings of the Distributed Ada '89 Symposium*, Cambridge University Press, Ada Companion Series (1990) pp. 177–199, Chapter Ada for Tightly Coupled Systems.
5. *DSP56000 Digital Signal Processor User's Manual*, Motorola, Phoenix AZ, 1986.
6. *TMS32010 User's Guide*, Texas Instruments, 1983.
7. *TMS32010 User's Guide*, Texas Instruments, 1985.
8. Simar, Ray Jr. The TMS320C40 and its Application Development Environment: A DSP for Parallel Processing. In *Proceedings of the 1991 International Conference on Parallel Processing, Volume I*, CRC Press, 12–16 August 1991, pp. 149–152.
9. Simar, Ray Jr., Koeppen, P., Leach, J., Marshall, S., Francis, D., Mekras, G., Rosenstrauch, J., and Anderson, S. Floating-Point Processors Join Forces in Parallel Processing Architectures. *IEEE Micro 12*, 4 (August 1992), pp. 60–69.
10. Stenström, P. Reducing contention in shared-memory multiprocessors. *Computer 21*, 11 (November 1988), pp. 26–37.
11. Smith, A.J. Cache Memories. *Computing Surveys 14*, 3 (September 1982), pp. 473–530.
12. Borrill, P.L. MicroStandards Special Feature: A Comparison of 32-bit Buses. *IEEE Micro 5*, 6 (December 1985), pp. 71–79.
13. Borrill, P.L. Objective Comparison of 32-bit Buses. *Microprocessors and Microsystems 10*, 2 (March 1986), pp. 94–100.
14. Edwards, R. Futurebus—The Independent Standard for 32-bit Systems. *Microprocessors and Microsystems 10*, 2 (March 1986).
15. Theus, J. Futurebus+ Parellel Protocol. In *Northcon/89 Conference Record*, 17–19 October 1989, pp. 329–334.
16. Sha, L., Rajkumar, R., and Lehoczky., J.P. Real-Time Computing with IEEE Futurebus+. *IEEE Micro 11*, 3 (June 1991), pp. 30–33, 95–100.
17. Sweazey, P. and Smith, A.J. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, 2–5 June 1986, pp. 414–423.
18. Cantrell, J. Futurebus+ Cache Coherence. In *Northcon/89 Conference Record*, 17–19 October 1989, pp. 335–342.
19. Motorola, *MC68030 Enhanced 32-bit Microprocessor User's Manual*, Prentice-Hall, Englewood-Cliffs NJ, Third Edition (1990).
20. *Microprocessors, volume II*, Intel, Santa-Clara CA, 1991.
21. *MC68040 32-bit Microprocessor User's Manual*, Motorola, Phoenix AZ, 1989.
22. Bryant, R., Chang, H.Y., and Rosenburg, B. Experience Developing the RP3 Operating System. In *Proceedings of Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, 21–22 March 1991, pp. 1–18.
23. Owicki, S. and Agarwal, A., Evaluating the Performance of Software Cache Coherence, report MIT/LCS/TM-395, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1989.
24. Owicki, S. and Agarwal, A. Evaluating the Performance of Software Cache Coherence. In *ASPLOS-III: Proceedings of the Third Internation Conference on Architectural Support for Programming Languages and Operating Systems*, April 3–6 1989, pp. 230–242.
25. Bolosky, W.J., Fitzgerald, R.P., and Scott, M.L. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Published as Operating System Review, 23(5), ACM Press, December 3–6 1989, pp. 19–31.
26. Hwang, K. and Briggs, F.A. *Computer Architecture and Parallel Processing*, McGraw-Hill, New-York NY (1984).
27. Weber, W.D. and Gupta, A. Analysis of Cache Invalidation Patterns in Multiprocessors. In ASPLOS-III: Proceedings of the Third Internation Conference on Architectural Support for Programming Languages and Operating Systems, April 3–6 1989, pp. 243-256.
28. Agarwal, A. and Gupta, A., Temporal, Processor, and Spatial Locality in Multiprocessor Memory References, report MIT/LCS/TM-397, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1989.
29. Gentleman, W.M., MacKay, S.A., Stewart, D.A., and Wein, M., Using the Harmony Operating System: Release 3.0, ERA-377, Division of Electrical Engineering, National Research Council Canada, February 1989.
30. Stewart, D.A., and MacKay, S.A., eds. Harmony Application Notes (Release 3.0), ERA-378, Division of Electrical Engineering, National Research Council Canada, February 1989.

31. Gentleman, W.M. Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept. *Software—Practice and Experience 11*, 5 (May 1981), pp. 435–466.

32. Gentleman, W.M. and Shepard, T. Administrators and multiprocessor rendezvous mechanisms. *Software—Practice and Experience 22*, 1 (January 1992), pp. 1–39.

Discussion

Question G. HAMEETMAN

Should you not put more emphasis on the fact that you problem only exists in a heterogeneous processor environment? Almost all modern processors have solved your problem for homogeneous system, eg Transputer with message passing model, R4400 with a separate cache snooping bus.

Reply

You are right in stating that most, if not all, microprocessors with cache provide some cache consistency protocol capabilities. However, the problem is in the way these components are used in real embedded systems, airborne or ground. The fact is that most embedded systems are build around the Motorola 68000 family, the Intel 8086 family and, to a lesser extent, the AMD 29000 and Intel i860. These processors are put in cards with some local memory and I/O devices and access other cards over a backplane bus. The most common bus is the VME bus, but the NuBus, the Multibus II and some other busses are also used. Typically, snoopy cache consistency protocols cannot be used across these busses because of the design of the bus interfaces.

The thinwire approach allows one to implement real-time multi-tasking systems on multi-processors built from such cards. It also allows one to port with no modification software components (written in a high level language) and entire subsystems (components and design) to machines with no shared memory, such as a Transputer-based multprocessor. The components I have in mind are tasks, such as server tasks and high level (task-based) device drivers, which could be written in Ada. The thinwire approach thus provide for code and even design re-use.

Question L. HOEBEL

Depending somewhat on multi-processor configuration, don't you somewhat overstate the case against snoopy cache coherency protocols and is the predictability of cache performance not just like translation (look aside) buffers for page tables? ie, can't you determine average performance?

Reply

When building real-time systems in a high level language such as C or Ada, one does not have control over the allocation of data to memory for static data and data allocated in stack frames; If any of these data is shared as happens when a variable local to an Ada task is passed as a parameter in all entry call, false sharing can occur. The implication of this, and the implication that snooping is controlled at the page level, is that snoop cycles can occur wherever a local variable is modified. This is the case with the MC 68040, which must be used in a write-through mode for snooping to work. Therefore, I do not think that I am overstating the case over using snoopy protocols, at least not for the type of multi-processor I considered, which is representative of a large number of real-time platforms.

Of course, the real cost of snooping in a given application depends on the underlying machine and on the application itself. Studies of the cost of snooping do exist. These studies looked at "typical" applications. Such "typical" applications could behave quite differently from a specific real-time application.

The in predictability introduced by cache consistency protocols is not the same as the impredictability introduced by translation lookaside buffers or by caches that are not kept consistent in hardware.The difference is that with cache consistency protocols, the state of a given cache in no longer a function of the addressing history of its processor, but also a function of the addressing history of all processors. Without cache consistency protocols, it is possible to determine what addresses will be generated by a processor, and therefore to know what the state of its cache (and TLB) will be, if the input data to the program is known (to determine the execution path) and if asynchronous interrupts do not occur.

Question C. BENJAMIN

Which message passing machine are you considering for your implementation?

Reply

A specific machine has not been selected. Rather, the investigation has so far concentrated on finding an efficient machine-independent protocol for interprocessor communication. Only then will actual machines be selected and the performance of the implementation measured.

On ground System Integration and testing : a modern approach.

B. Di Giandomenico
AIDASS Responsible
ALENIA DVD TEST
Corso Marche 41 10146 Torino Italy

1 Introduction

Modern aircraft, military or civil, are incorporating all the most up-to-date technology in all fields of human sciences. There is an increasing tendency to develop digital control systems which are rapidly replacing analog control systems in all areas, especially in the traditional ones such as engine control, power generation, fuel and environmental control etc.

Digital systems are already acknowledged as unreplaceable in avionics and are fast growing also in flight control systems. Indeed they are responsible for the increased sophistication of modern aircraft and for the birth and success of avionics as we now know it.

The net result is that where in the first generation of jet planes there were no on board computers, modern aircraft may have more than twenty, with single or multiple 32 bit microprocessors, multiple megabytes of RAM and sophisticated real time operating systems.

Figure 1 may be a good example of the tendency portrayed before, with the trend clearly highlighted for future aircraft.

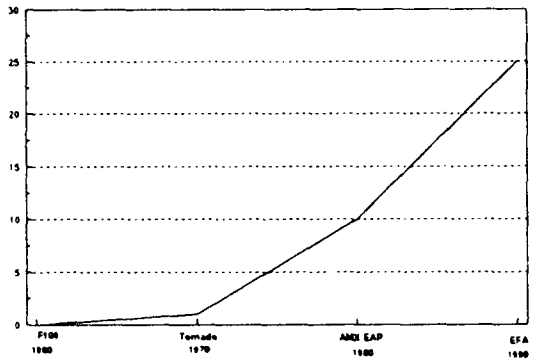


Fig. 1 - Number of on board computers

Aircraft interiors have dramatically changed in the last twenty years; a modern combat aircraft is mostly an empty shell with lots of space for equipment mounting and there is very little resemblance to older mechanical aircraft. Every computer for which we find a place is something which can augment the aircraft capabilities. This is much more felt in combat aircraft (where space is at a premium and every cubic inch counts) than in civil aircraft.

As a result also aircraft design has changed, calling in professions which were not present in the past. Where aeronautic engineers were predominant in the past, they are now just a part of the whole design cycle; the presence of electronic engineers and software engineers is now growing and as systems get more complex shall become the predominant category of professionals engaged in the design. They are the people who design the mission capabilities of the aircraft, and in the case of unstable airplanes they are also the people who keep the aircraft flying. Software, as already written in many presentations, is fast becoming the most expensive activity and the longest in terms of application tuning.

System design and testing can be briefly described with the fall model depicted in figure 2 which is also already known in different forms and which is also widely criticised as being oversimplified. In this instance the author only wishes to use it to highlight the area of the system development which is the target of this presentation, namely the hardware to software integration.

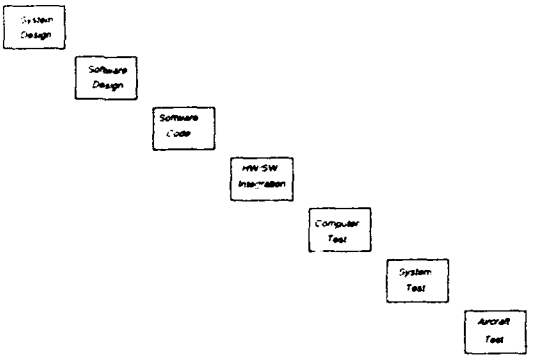


Fig. 2 - Aircraft prototypes design phases

2 Task definition

Hardware to software integration definition is defined as that activity whose aim is the marrying up of the software previously host tested with the equipment freshly manufactured by a supplier. Output of this activity is therefore a combination of a reasonably bug free computer and of a software load reasonably tested and certified

The activity can be split in three main areas :

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

- familiarisation with the equipment;
- static testing;
- dynamic testing.

3 Areas of work

3.1 Familiarisation

This is that part of the activity which is more akin to art than to science; past experience suggests that a hardware man with knowledge of practical hardware (usually the equipment engineer) and of instruments for debugging (i.e. emulators and usual laboratory stuff) and one or more software people make up the best team to tackle the unknown quantity. The first tests in downloading and running part of the software usually fail miserably, so it is time for the hardware man to delve in the deepest recesses of the computer using the emulator and maybe the logic state analyzer in order to discover why that interrupt is not high, what is contained in those two bytes on top of the memory, what is actually written in the chip registers etc.

3.2 Static testing

As soon as the team succeeds in running a few example programs on the target hardware, it is time to start downloading also part of the software which is to be officially tested and it is time also to start testing program stubs which stimulate the hardware input/output, in order to understand better how the driver software works (if it works at all), how long it takes to set a discrete output or to read a digital input, and how this affects the global scheduling.

Last but not least, measuring the time it takes to the program to run is also apt to deliver nasty surprises, with the possibility that the program, as it is, might not fit inside the scheduled time.

This area of testing also involves the setting of simple sequences to be executed as input to the target computer and the recording of the outputs. Many tests can indeed be performed with these simple guide-lines.

3.3 Dynamic testing

This area covers all the testing which is performed by implementing a closed loop test environment, between the target computer and a test harness capable of stimulating, monitoring and recording all the data transactions. This is the area which has lately become the one most in need of growth in terms of test harnesses and is the one which shall the subject of this paper, illustrating how one of these systems has come to be, how it has grown to be an invaluable tool in performing the hardware to software integration and system integration and testing.

4 AIDASS

AIDASS stands for Advanced Integrated Data Acquisition and Stimulation / Simulation System. It is an acronym which sprang up during the first years of the EFA project (in the Tornado years everybody had a DASS, with EFA it has become advanced and integrated) and has been applied with some slight modification to the data acquisition system of the four EFA partner companies.

5 Some history

Our job started in 1987 as a project for a Tornado data acquisition system, whose specifications were very simple:

- the system had to acquire a number of data, namely analogue and digital;
- it had to act as 1553 bus controller or bus monitor and at the same time simulate the missing remote terminals on two buses;
- it had to simulate a number of missing equipment which talked over a Panavia Standard Serial Line.
- it had to record the activities performed in order to be able to get a report of the test or to be able to analyse any malfunction.
- of course it had to be user friendly, easy to use, modular for an easy future expansion.

A first analysis of these requirements was coupled with a very simple analysis of aircraft architectures, in the electronic department: these can be divided roughly in three areas, general systems, avionics, flight control systems. These systems have fairly different layout and needs :

1) general systems have a predominance of digital,
- analog, frequency etc, signals over copper, looms of wires and one or more buses (which can be 1553 or else) over which a small traffic runs;

2) avionics see a predominance of buses with
- heavy traffic, and little if nothing at all in the analog and digital signal field;

3) flight control systems have a mixed
- environment, more similar to general systems, but with very stringent timing requirements and a strong need for closed loop simulations.

We had to design something which was general enough to be able to cope these systems. This means a high modularity, with the possibility of inserting pieces as needed, pieces which can be high speed simulators, analog control boards, bus interface cards etc. There is a need for a number of I/O monitoring, but it can be safely restricted to a few hundred points, not thousands. The rate of the system has to be as close to the aircraft's

On ground System Integration and testing : a modern approach.

B. Di Giandomenico
AIDASS Responsible
ALENIA DVD TEST
Corso Marche 41 10146 Torino Italy

1 Introduction

Modern aircraft, military or civil, are incorporating all the most up-to-date technology in all fields of human sciences. There is an increasing tendency to develop digital control systems which are rapidly replacing analog control systems in all areas, especially in the traditional ones such as engine control, power generation, fuel and environmental control etc.

Digital systems are already acknowledged as unreplaceable in avionics and are fast growing also in flight control systems. Indeed they are responsible for the increased sophistication of modern aircraft and for the birth and success of avionics as we now know it.

The net result is that where in the first generation of jet planes there were no on board computers, modern aircraft may have more than twenty, with single or multiple 32 bit microprocessors, multiple megabytes of RAM and sophisticated real time operating systems.

Figure 1 may be a good example of the tendency portrayed before, with the trend clearly highlighted for future aircraft.

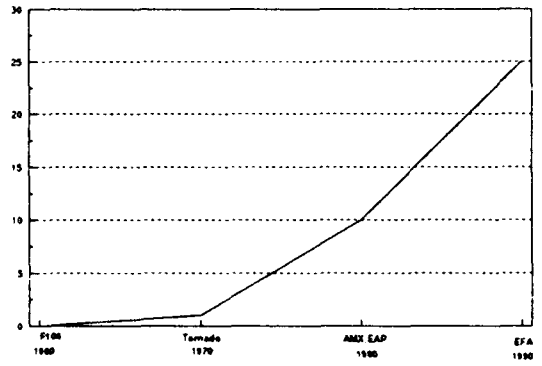


Fig. 1 - Number of on board computers

Aircraft interiors have dramatically changed in the last twenty years; a modern combat aircraft is mostly an empty shell with lots of space for equipment mounting and there is very little resemblance to older mechanical aircraft. Every computer for which we find a place is something which can augment the aircraft capabilities. This is much more felt in combat aircraft (where space is at a premium and every cubic inch counts) than in civil aircraft.

As a result also aircraft design has changed, calling in professions which were not present in the past. Where aeronautic engineers were predominant in the past, they are now just a part of the whole design cycle; the presence of electronic engineers and software engineers is now growing and as systems get more complex shall become the predominant category of professionals engaged in the design. They are the people who design the mission capabilities of the aircraft, and in the case of unstable airplanes they are also the people who keep the aircraft flying. Software, as already written in many presentations, is fast becoming the most expensive activity and the longest in terms of application tuning.

System design and testing can be briefly described with the fall model depicted in figure 2 which is also already known in different forms and which is also widely criticised as being oversimplified. In this instance the author only wishes to use it to highlight the area of the system development which is the target of this presentation, namely the hardware to software integration.

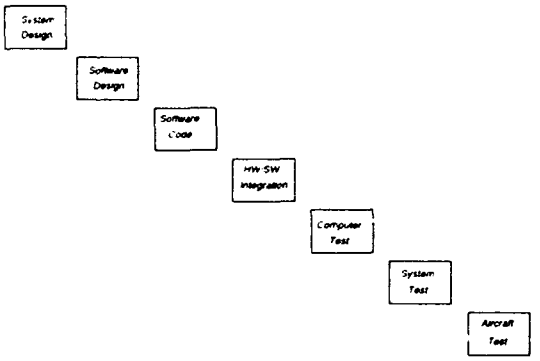


Fig. 2 - Aircraft prototypes design phases

2 Task definition

Hardware to software integration definition is defined as that activity whose aim is the marrying up of the software previously host tested with the equipment freshly manufactured by a supplier. Output of this activity is therefore a combination of a reasonably bug free computer and of a software load reasonably tested and certified

The activity can be split in three main areas :

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

one as possible, which for our aircrafts usually means around 50 Hz, with the possibility of including high rate parts.

In order to fulfil these requirements a market survey was conducted to learn of the current state of the art of the data acquisition systems and whether they could be the equal to the task. Unfortunately all the 'common' data acquisition systems we surveyed fell short in some fields; usually they were very capable in their chosen field with capabilities of million of points acquired in one second, PCM, data reduction and some were even capable of housing simple simulation to stimulate the target computer. Usually they could not do all together, and as a certainty they could not act as closed loop machines, not unless their architecture changed drastically.

So we were in the position that we had to devise something of our own to fill in our need.

6 Start of AIDASS

6.1 Birth of an architecture

The first problems we faced regarded the type of the architecture to be used; did we have to go for a centralised system or for a distributed system ? What kind of I/O cards would we be using ? What machines would we be using ? What operating system ? and so on ...

An additional market survey was conducted to learn about computers and I/O systems, and as a result an embryo architecture started to appear. VME in those days was starting to rise powerfully as I/O subsystem, so we almost took it for granted. From the software point of view then, our historical environment has always been a **DIGITAL VMS** one, with no ties except for sporadic contacts with the **UNIX** world. As **VMS 5.0** and **DECwindows** had just been announced, it also seemed natural to turn to them in order to capitalise on the internal software expertise on VMS and to build a graphical user interface based on DECwindows in order to build on a known international standard as X-Window. We had some of the pieces and we had to tie them together : a user interface, I/O cards and CPUs to control them. One of the choices had been made already, we were going for a distributed processing system. **ADA** would be used as far as possible for the same reasons as VMS and also for practical reasons, (it was going to be the language used for the writing of the simulations by our partner companies, so if we wanted to be able to share something with them, we had to design something able to accommodate ADA programs). VAXeln was also chosen as the only way to run VMS ADA programs in the easiest possible way. What we were still missing was a way of shuffling the data to and from the parts comprising the system and to connect them. One of the methods we looked for was a many ported ram acting as glue. We didn't actually found one in the first round, but we went close enough, choosing a bus adapter from QBUS to VME with dual ported ram between the buses. The first **AIDASS** was born. Its architecture was as illustrated in pic. 3), with a VAXstation 3500 acting as user interface and real time recording, a RTVax 3200 which was to host all the simulations and a VME subsystem for the I/O. It was easy to find a VME bus extender for those occasions where one crate was not enough to host the I/O cards.

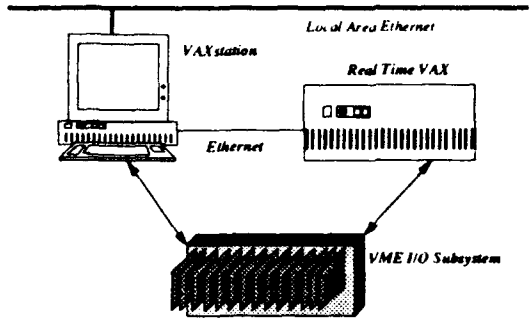


Fig. 3 - First AIDASS architecture

6.2 Software

Well, the birth of the architecture was painless enough, now what about the software ? DECwindows was something so new that nobody knew much about it, much less how to use it in conjunction with ADA. In those days (remember that we are talking about 1988) a proper programmer used C with X-Window (90 % still do it today), but luckily we had some examples provided with DECwindows which helped us start. That was one task, together with all the underlying software to handle the many tasks of the user interface.

Another one was to think of the interfaces with the external equipment, with the VME in the first place, how to program it and how to interface with it. Having chosen VMS we also looked at a compiler for the VME boards. The best processor we were able to find were Motorola 68020 but ADA for them was not the best option. We then chose C as language for the VME cpus, with the pSOS^{+m} operating system kernel to act as scheduler.

VAXeln with ADA for the RTVax was another task, with no conceptual ground to break and not technically difficult.

The basis of the system anyway is not in the software but in the underlying database, which contains all the information used to run the tests. The database is really what gives to the AIDASS its capabilities, because it contains an adequate description of everything on the rig or bench, a

complete interface description of all the interesting input/output of the equipment present on the rig with names, engineering units, scaling informations, and in the case of bus signals also transaction tables, subgroups and bit/byte/word position and LSB and MSB. The database is as complex as it looks, but for us it would be a by-product of the engineering activity, because such an ICD database has to be compiled by the engineers for the EFA activities. It uses the INGRES DBMS and we get the data from it, adding just those information needed to link the logical signals of the computers with the interfaces present on the rig (this means telling AIDASS which computer signals went to which interface card).

6.3 Experience

To design the system software we adopted the standard tools used for aircraft software design, **CORE** for requirement and **HOOD** for software basic and detailed design. In particular this latter technology was particularly suited to the event driven X-Window environment, while we had some problem to describe the same environment using the procedural **CORE.** It was done nonetheless. After 6 months of coding finally the first product was ready to be used. During the coding phase many were the problems tackled and solved by the engineers; among them the toughiest had to do with DECwindows. Implementing a user interface using X-Window proved to be a lengthy process, with many revisions to each mask to get the right position of the text, the right font, attention was put on not superimpose words and so on. All in all a skilled developer could not achieve more than 2 or three masks per day when they were simple, just because of the tediousness of the process. Today there are tools which enormously simplify this design phase and a skilled developer can churn out ten complex masks per day easily, testing them in real time with facilities given by the tool, whereas we were compelled to write program stubs to test the mask.

On the VME side there were the usual skirmishes during the familiarisation with VME CPUs, the problems in understanding how interrupts were generated and how we could trap them with our operating system kernel to provide the scheduling required.

A continuous feedback with the users was needed to solve some points when the software designers didn' t know how to best proceed and many changes were made to emphasize user friendliness and the general usability of the system. A set of tools were developed to take care of the database which more and more came to represent the central part of the system. Database creation and population tools and the very important tool of database consistency checking came slowly into existence. Consistency you had to have in order to avoid two or more signals sharing the same output pin, or the incorrect scaling of some data (a boolean entity must not have more than two meanings, analog signals must have limitations etc.).

6.4 The progress

The progress was slow, much more so since the compiling and testing of the ADA program took more and more time, with unforeseen side effects of using DECwindows slowly creeping in. Also the documentation was not totally satisfactory, sometime ambiguous, and a number of bugs in the X-Window server were uncovered, some were circumvented and patches were delivered by Digital. The application was getting very big, totalling almost 200000 lines of code, between ADA, C, and UIL. The real time recording took some fiddling with VMS, but eventually it all started to work.

6.5 The result

What we got in the was a system which was able to perform tests automatically, defining for each test the variables which had to be recorded, those which had to be stimulated and how (with a limited library of simple stimuli such as sawtooth, ramp etc), and what had to be recorded. All of these activities were performed by clicking with the mouse on appropriate menu and lists. Then the test was started and the data were shown in the chosen engineering units format, with either an optional graphical representation or signal true input/output format could be displayed in parallel. At the end of the test, the data collected could be analysed using some embedded facilities which allowed to plot the variables in time plots and allowed some matching with other variables. During the test the input variables could be stimulated at run time by clicking on the variable and defining the type of stimulus, its length and amplitude.

Now the time had come to use the AIDASS for a real situation, connecting it to a real on board computer. Time to populate a real database with real aircraft data, check it and then see what came out of the I/O boards.

6.6 Under operation

After a few days of working with the database tools, one thing was clear: we had to have a better tool, the one we had now was too complex and time consuming. The insertion of the data was riddled with difficulties and too often mistaken data was inserted with no help from the system, except at check time when an avalanche of errors were detected by the system. In addition many small difficulties were experienced with the user interface, which is only understandable as the product was brand new out of the developers' hands. With real data we also found that system

initialization time was far too high when an high number of data was inserted (it might be as high as half an hour). Also the checking process was found faulty and errors were found during the test run due to inaccurate checking of the database.

All of these teething problems were investigated and solutions were proposed to the users. It took more than one round to get most of the problems ironed out.

The simulation part was easily the most powerful part of the system, because by using the variable names defined in the system database, a program was able to access every resource in the system and change it at will. A few difficulties were met at first with the definition of the interface between the simulation and the system and that was ironed out by the development of a tool which automatically extracted the interface from the database and built it.

6.7 The second generation

While we were happily developing our software, the external world started to change dramatically. Digital did not manufacture the QBUS workstations any more, and we were stranded with an architecture which was heavily based on QBUS. We had to start developing something like ten more of these AIDASS nodes and clearly we could not afford to find again ourselves in this situation. All of this prompted us to think of ways to avoid the occurrence of these problems again. We performed a second market survey looking for hardware and what we found was disconcerting: workstation manufacturers were steering clear of I/O buses as much as possible, the market was moving toward busless workstations which were low-cost but which were clearly not well suited to our task.

We had to take an evolutionary step, while trying to save as much as possible of the software written. Based on the experience done with the first node, the first thing was to decouple all real time activities from the non strictly real time. Second prerequisite was the modularity of the system, which was to be retained at all costs. These requirements led us to rethink also about the other components of the architecture and to clearly label them as separate components if possible, in order to allocate for them a separate piece of hardware.

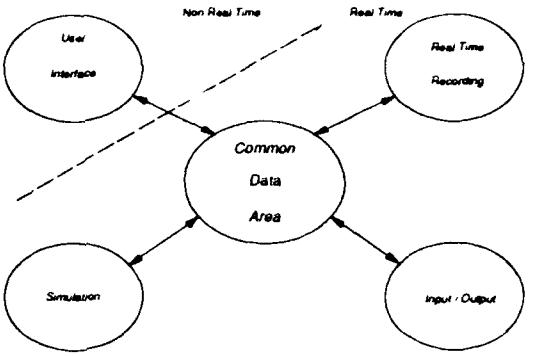


FIg. 4 - Logical AIDASS architecture

According to our requirements we built a logical architecture depicted in fig. 4., separating the real time part from the non real time. Then we tried to find corresponding pieces of hardware to transform the logical into physical. Some of the pieces we already had, such as the VME, the simulation part, and the user interface could be reused almost totally. What we lacked mostly was the common data area and the link between it and the user interface. Many alternatives were examined, and in the end discarded because it meant tying ourselves to a specific bus on the user interface part, so we reached the decision of totally uncoupling it from the rest of system, by connecting the VAXstation to the real time using Ethernet. On the VME side there had been also a development with the advent of the RTVax chip on VME boards, and it was easy to put one of these new RTVaxes to handle the communication between the two pieces.

We had a new architecture, and what was best we had saved almost 90% of the software already written, we only had had to rewrite part of the interface software.

What came out looked even more impressive than the first architecture, actually costed much less and was highly flexible.

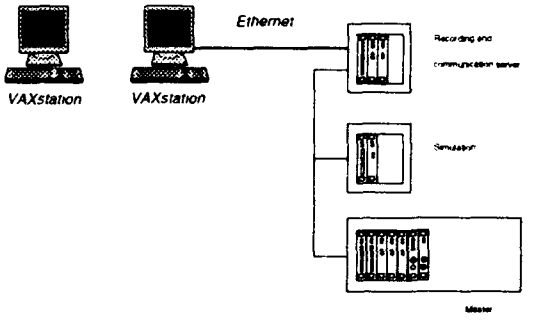


Fig. 5 - The new architecture

6.8 New experiences

This new architecture was soon put to the test on a couple of benches, while the first one was finishing its tests on another bench. The new one proved soon to be much better than the older one, thanks also to the fact that more powerful hardware had been employed as workstations and VME cpus, but still bugs crept in, especially in the 1553 handling. We had a very tough 1553 bus to handle and it was very hard to describe the transaction in a complete manner, both because it was very long and complex and also because it had a complex structure with multirate signals on the same subaddress and a very high data bus load.

The database went under a couple of major changes with the addition of a few data fields in order to accommodate the most generic bus of all and with it went also modifications of the database check tool which by now had been speeded up of an order of magnitude. After the first grumbling, the test people were now quite happy of this new tool as they appreciated the breadth of possibilities now available. Thanks to X-Window, the setting up of test databases was very easy, boring, because you had to choose from a very long list the name of the variables you wanted to handle and how. The running of the system was quite easier now after many bouts with the users and as a consequence real work could be now started and the tool depended upon for its results.

Yes, this was the turning point of all our activity, the fact that the tool could be safely trusted to give the right results, so that it could be used to diagnose the system health, and thanks to the possibility of dynamically simulating the missing equipment in all of their functionalities, to anticipate on the HW/SW benches sime of the tests related to the system.

6.9 Other developments

Once the software stabilised, other capabilities have been added, expanding upon the visualization of the data in engineering units, in order to give the user the maximum flexibility; an advanced error injection was developed to allow further debugging of the systems under test, both from the discrete/analogue side and the bus side. An optimization was performed on the software so as to wring every ounce of power from the cpus and as a consequence now the system is capable of supporting very high loads of I/O intensive rigs. Faster than the base rate simulations and data acquisitions are easily handled and inserted painlessly in the system, with rates going now up to 2 Khz per channel. Multiple cpus, even RISC cpus, running each a simulation program can run concurrently as long as the writer is careful about the data they exchange and manipulate.

The limitations are still there, though, and we still do <u>not</u> have a powerful data acquisition only system, it still is less and more than that, a mixture which best caters to our needs.

A spin-off of this adventure has been the recent porting of the simulation software module base under VMS, to allow for the testing of simulation software under VMS. The software will run not in the real time, but will be clocked as under real time, allowing a thorough debugging of the modules and interfaces.

This idea is now being expanded upon and we are investingating ways of exapnding this methodology also for software / software integration on the host, for the final part of it at least to allow for an easy transition between this phase and the hardware / software integration phase, with a consistent tool and interface. This is being studied into right now.

7 Conclusions

This is a short history of how a complex data acquisition system has come into existence, to satisfy the needs of modern aircraft testing during the hardware software integration and integration testing.

What we now have in our hands is a complex and composite tool, capable of supporting testing from the last stages of software / software integration till the end of all the integration testing on a system rig.

It is still changing and adding in purposes and in supported hardware to cater for all needs envisaged in our work.

We believe that it shall be more than good enough to reduce the workload on the test engineers and to allow them to better analyse the equipment under test as required by the complexities of modern computers and their embedded functions.

Discussion

Question D. NAIRN

If you were starting all over again, knowing what you know now, what would you do different?

Reply

I would probably go for more open systems with Unix workstations and Unix supported targets on VME (Lynx-OS, VX-Works, etc). I would not change the architecture because I think that for the time being, it is the most functional for the tasks it has to face, namely to handle inputs and outputs to/from the target. The system has to acquire data, think upon it and react, a closed loop situation. In addition, I would like to scrap Ada, because it is highly not portable among systems, I would better use ANSI C.

SOFTWARE TESTING PRACTICES AND THEIR EVOLUTION FOR THE '90s

Patrizia Di Carlo
Alenia-Finmeccanica S.p.A.
Corso Marche 41
10146 Torino
Italy

SUMMARY

Experience within an aerospace company on solving specific problems in the host testing of embedded software is described.
Operational applicability and exploitation of tools off-the-shelf, solutions for the management of both process and testing information, and, finally, the impact of the possible use of formal methods for the automatic generation of test cases are investigated and assessed from the user point of view.
The driving topics are the analysis of the effects of the assessed solutions on the current working practices together with their transfer to operational divisions.

1 INTRODUCTION

AIMS (Eureka Project n. 112) is an industrial research project whose primary objective is prove the suitability of proper software technologies to provide adequate solutions to a number of problems common to the aerospace community.

An initial *Definition Phase* identified several categories of problems currently affecting Aerospace companies when developing an Embedded Computing System (ECS). Starting from this basis, during the subsequent *Demonstration Phase* the AIMS Project focussed on specific areas of the life-cycle to highlight the nature of the problems and their direct and indirect consequences.

Four demonstration projects were started, each of them following the same problem-driven approach: their goal was to determine the impact of selected technologies to solve specific problems identified in the previous phase. Expected benefits of applying these technologies will be measured to provide acceptable evidence and to assess their applicability in real system development.

Assessment will be performed in terms of quality, costs, time-scale and co-operation benefits. These results are expected to enable future aerospace projects, that intend to use the improved practices or support technologies, to take a decision about their adoption on quantitative basis.

Each assessment will mimic a real development environment dealing with information from real projects - typically a sub-system from one of the various aerospace projects recently undertaken by the companies - and it will directly involve the practitioners from the specific application domain. The effort spent during assessment projects will have an immediate pay-back for all partner companies since they will be able to exploit single successfully assessed technologies on current projects, even before an integrated AIMS solution is provided in the last phase of the project.

The Alenia Demonstrator is being carried out by Alenia Divisione Velivoli Difesa (Defence Aircraft Division) and focuses on improving practices in the area of software module testing and software/software integration testing.

2 THE PROBLEM DOMAIN

Software testing is an expensive and time-consuming activity in software development projects within the aerospace domain. This is demonstrated by the fact that a large percentage of the overall development effort is spent on testing activities: this percentage ranges between 25% and 50% depending on the project size, complexity and criticality.

The huge effort spent in testing, however, is not paid back by a sufficient increase in the quality of the final product. This is clearly not due to a lack of effort but to the lack of a systematic approach to software testing. Furthermore, while the management looks at testing as an expensive activity, technical personnel consider it as particularly boring. In addition, it is often the case that software testing is sacrificed when the project is behind schedule.

A detailed analysis of the way of working within different software teams has confirmed the current lack of methods and tools to adequately support testing activities. In particular, a number of crucial problems have been identified such as: lack of support for traceability (from requirements to test cases specification); lack of methods and tools to verify test completeness and effectiveness; the test case implementation is performed manually which implies considerable time and effort as well as higher

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

probability of introducing errors and difficulties in maintainability; poor support for test reporting and documentation. This list of problems revealed a great need for improvement of the current practices and a greater tool support.

The existing testing policies and methodology standards provide some criteria on how to perform and manage testing activities; however they define generally high level guide-lines about what items are to be produced, which test criteria must be used depending on the classification of the system under test, and so on. Each company has to tailor these directives to produce detailed company standards that are often badly implemented within the companies.

Nevertheless, the testing design remains mainly an intuitive process, dependent not only upon the selected testing method but also upon the experience of the testing team. It is still a problem that few developers have been educated and trained in testing techniques.

As far as testing tools are concerned, it can be said that very few are available on the software market compared with the wide offer of tools for other software development phases (requirement analysis, software design, coding). Furthermore, none of these tools have a large user base, and - in the Aerospace Companies - no great experience in the use of computer aided supports has been found.

It is clear that any increase in the effectiveness of the testing process turns into an improvement in the quality of the final product and that any increase in process efficiency results into an increase in development productivity and into a considerable saving in the overall project cost. In fact, a more reliable test process also implies a greater rate of errors detected before the start up of the system with a consequent increase in safety confidence and decreased maintenance costs.

3 SOFTWARE TESTING TOOLS

The achievement of a more cost-effective testing process is mainly based on the application of a rigourous testing method and the use of tools supporting the user during the most repetitive and/or automatable actions.

The use of testing tools can drastically reduce the global effort and, at the same time, it can provide the user a better understanding of the performed tests. This will bring some advantages such as the reduction of the manual involvement of the staff, an increased confidence in the quality of the product and an enhanced testing productivity. On the other hand, a greater emphasis should be put on education/training in the testing approach.

In recent years, an increasing number of tools supporting software testing have been introduced on the market. Currently available testing tools do not provide an exhaustive solution to the whole testing problem, but they may bring considerable benefits to the current practices by supporting a more systematic approach to software testing.

A first step toward a more intensive utilization of tools in the testing process was the analysis of the commercial of the shelf testing tools; this analysis was mainly oriented to the tools supporting test of Ada programs and that are available on VMS platform. A preliminary screening of these tools has been performed based on commercial documentation, technical literature and demonstrations.

The first result of this analysis is that only some aspects of the testing process are tool-supported (such as complexity measurement or coverage analysis) while some others critical phases (such as test cases specification, data selection, regression testing, tracing, reporting, functional coverage) are manually performed.

Particularly the higher part of the testing activities related to the identification of the functionality to be tested from the Requirement Specification is still mainly based on the tester experience and competence.

The management of the testing process and items is another critical factor that would require the adoption of good testing practices, the provision of guide-lines and monitoring through the testing process, and a good organization and consistency of all the test items (test cases, harnesses,...).

4 ALENIA DEMONSTRATOR

The thorough analysis of users' needs described above, pointed out that inside the Aerospace companies there is a demand to improve the testing practices and to increase their computer-aided support helping to "simplify" the activities, making them more rigourous but, at the same time, less "boring", automating (re-)testing as much as possible, and improving the management of test process and of the related information.

This is expected to have a double positive effect. On the one hand, by increasing the support in the execution of clerical work, it will relieve technical personnel from the most boring activities, thus improving the quality of their tasks and consequentially their satisfaction. On the other hand, it will ease the management of the whole process, with clear advantages for a crucial part of the development life-cycle.

Within the AIMS project, the Alenia Demonstrator aimed at identifying and evaluating a methodology to perform effective software testing in an efficient way. Three main areas where major improvements are expected were identified:

- the exploitation of a set of existing commercial tools that support specific phases of the software testing process, the assessment of the impact of their use in the current practices, and - if successful - their introduction inside the company;
- the development of a prototype of an assistant tool managing the whole testing process and supporting the tester during all testing activities - specification, implementation, execution, evaluation and reporting - as well as handling heterogeneous information that has to be produced and maintained throughout the testing process;
- the assessment of a formal approach to test case derivation starting from formal specifications to help automate such process.

These areas of research address some of the most crucial issues in software testing and contribute from different perspectives to the assessment of the possible improvements of the testing process. The related solutions will then be integrated into a coherent whole, covering both methods and tools.

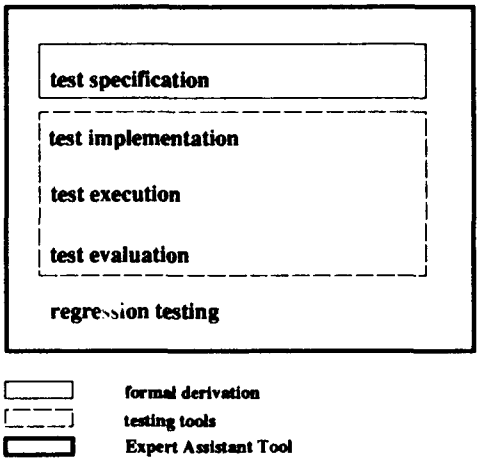


Fig.1: Demonstrator areas

The assessment of the Demonstrator is based on sub-systems from EFA and Tornado projects and it will provide proof that the investigated solutions do solve the specific problems.

The extra benefits to the current practices will be described in terms of improvements both in the productivity and in the product quality.
Metrics collection shall be performed during the whole demonstrator assessment: these measures will mainly take into account the *effort* required to perform the different testing activities and the quality of the tested products, in terms of early identified *errors*.

4.1 Exploiting Testing Tools

A preliminary evaluation of the state-of-the-art of the tools supporting the testing of Ada software was performed providing a first set of *candidates* that could be introduced in current practices. Each of these tools was evaluated - on real case studies - and compared on the basis of a grid of desirable characteristics and considering the effect of their introduction in the current testing practices. Some of these tools were already available inside the company but not currently or appropriately used.

The candidate tools may be divided into three main classes according to their functionality: static analysers, test cases specification tools, coverage analysers.

Static analysis consists of the analysis of source code - without the need for any executable form - to determine properties of programs which are always true for all the possible execution conditions. Specifically the purpose is to verify the coding standard conformity and to measure complexity. The most sophisticated tools can also check the use of each single data, find out unreachable code and produce cross reference of functions and data.

Test cases specification implies the selection of the unit to be tested, the possibly needed stubs (simulation of the called units), the definition of the test driver, the selection of the input data and the definition of the expected output. Test specification tools can help the tester in these activities giving mechanisms to implement test drivers in an efficient and structured way, defining generalized stubs, keeping trace of tests execution and producing tests report. The available commercial tools in this area are either based on a test definition language, or on a set of functions embodied in a programming language.

Coverage analysers monitor the execution of tests and produce a report specifying which parts of the software under test have been exercised using particular coverage criteria (statements, branches, ...).

To assess and compare the different tools (fig. 2), some evaluation criteria have been defined. They include some general aspects of the tool such as supplier/vendor, assistance, installation, user friendliness (both with respect to learning and use), performance, hardware/software environment required, and so on. For each class of tools relevant characteristics have been identified which are related to the specific functionality, and to the capability to automatically produce documentation and test reports.

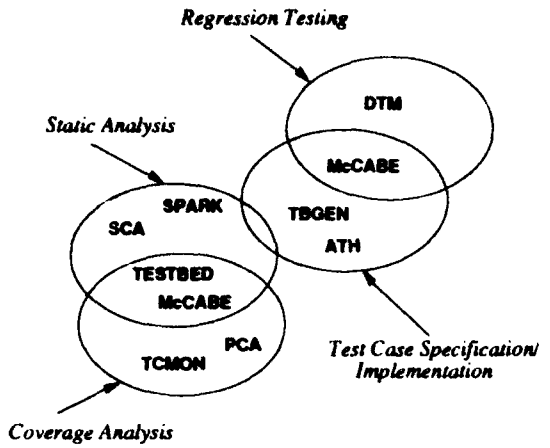


Fig. 2: Evaluated tools classification

Useful indications about their benefits and drawbacks have been reported and a subset of the tools has been identified to activate an experimentation on real case study. This experimentation led to the conclusion that some testing tools are mature enough to be used in real projects and even if they do not give a complete support, it seems that they can lead to relevant improvements.

After the evaluation inside the Alenia AIMS team of the most interesting tools, some tighter cooperation with the development teams was experimented and gave positive results; two tools - TESTBED (static and dynamic analyser) and TBGEN (test specification tool) - were planned to be transferred.

The first step of the technology transfer was the setting up of the testing facility and environment inside the testing team, were the selected tools have been installed; then the transfer to the testing team have been activated by means of training, tutoring, exercises always supported by the AIMS team. Such steps require not only qualified resources to manage the transition, but also strong management commitment and may have an organizational impact.

However the techniques supported by the tools do not require specific basic education and can be fairly easily mastered by average software personnel. Nevertheless development teams are likely to exhibit a variable degree of resistance to change, especially where a tradition about how to implement and execute tests already exists or custom/project tools and utilities have been used for a long time.

4.2 Management of the testing process

Test support is a crucial problem that concerns the management of a vast amount of information that is generated, accessed and manipulated throughout the testing process, ranging from test specification documents to software units, test drivers, reports and so on. It also involves the definition of appropriate methods - as a way to perform activities - and their combination within a testing methodology.

The Demonstrator that the Alenia team developed in order to cope with these problems consists of a prototype *Expert Assistant tool for Testing (EAT)*, which addresses two main aspects:

- the management of *testing information*,
- the management of *testing process*.

The purpose is to capture the knowledge required to perform all testing activities and to manage all products generated by such activities. The expected benefits are: first to improve testers' productivity, both by increasing the confidence in what they are doing and by providing inexperienced testers with a systematic way to perform testing; second to improve the product quality.

The EAT prototype architecture includes an object management system, a knowledge base and an advanced human computer interface (fig. 3). It has been implemented on VAX/VMS with OSF/Motif using commercial tools such as a relational database management system and a moderately complex but quite efficient expert system shell for the development of the knowledge base. This shell benefits of a rule based language which provides adequate means to incrementally develop such a system.

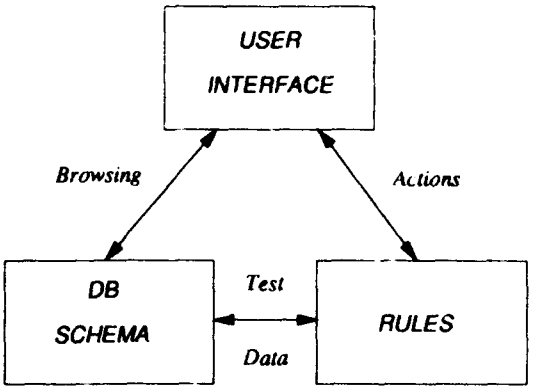


Fig. 3: EAT prototype architecture

The different aspects of the knowledge related to the testing process model have been implemented by means of sets of correlated constraints on the activities (rules). A first set of rules represents *integrity constraints*, that enforce the consistency of related objects in the database; a second set includes *process support* rules, which refers to how objects can be used and produced by activities. Finally, there is a third set of rules that may implement a given testing *strategy*, which can be defined at a project or organisational level: these rules shall be inserted in the knowledge base to tailor it to more specific environmental requirements. For instance, the

strategic rules may enforce a particular order during the test of the software units (top-down or bottom-up) or drive the user to perform a white-box testing with a complete structural coverage instead of a functional black-box testing. Strategic rules are related to specific project constraints (e.g. criticality classification or contractual requirements) and must be highly tailorable.

Tailorability of the knowledge base is only one aspect of EAT configurability. In fact, the system can be interfaced to other tools that have to be used during testing (such as editors, requirement specification tools, compilers and so on) according to user preference.

4.2.1 Information Management

The testing information management aspect concerns the storage and retrieval of all test items and the provision of an efficacious way to understand their relationships and navigate among related items. The objects under test are seen as a net of software items (each software item has some dependant units and depends from some others units) and requirements (relationships between software items and requirements). This aggregation allows the user to better understand the structure of the software under test and find the functional capabilities related to each software unit.

In a similar way, the produced test items consist of drivers, stubs, input and output data, and so on. These items are stored into the object base with the links between them and with the objects under test.

Therefore the user must be given a means to access and browse such information in a quite flexible way, assuming that during different phases of the testing process (from specification down to evaluation) different test related pieces of information will be relevant.

The functionality provided by EAT to manage test information are reproduced and briefly discussed in the following:

storage and browsing of test items
test items imported from a configuration management system or from other parts of the user environment, or generated while using EAT, are stored into a proper object base together with all related information (documents, source code, scripts) and linked with other relevant items.
The user can browse both items and links by means of an advanced human computer interface where several browsing windows can be opened, one for each type of test item (e.g. units, specifications, drivers). After selecting a test item in a browser window, the user may ask to browse all connected items, which will be shown in other browser windows, and so on. This allows the user to display all the information that is regarded to be relevant for the current testing activity.

editing and viewing facilities for test items
the user can also display the textual or graphical information that is directly associated with each test item: after selecting an item on a given browser (e.g. an input file for a test), the user can either read or modify it using proper menu choices on the browser. Obviously there are some pieces of information, such as requirements documents or source code of software units under test, that cannot be modified by the tester and whose browsers do not provide any editing option.

informal annotation of test items
during various testing activities the user is often willing to associate informal notes with different items (e.g. to record test design or implementation decisions or to annotate for future use): in common practice, this is done by hand-writing on documents or by using post-it stickers. The system allows this kind of information to be attached to each item by the choice of proper operation offered by the browsers' menus: in the current version only textual information can be attached, but provisions have already been made to associate graphical information.

coverage reporting
to give the user the means to verify the completeness of the performed tests, some reporting functions have been provided; by means of these utilities the user can see the percentage of the units and requirements already tested, their list and the status of completeness of their related tests.

import / export capability
Eat provides a way to collect all the pieces of information related to the software under test (i.e. requirements, software units, and traceability information) and store them into the object base. For this purpose the EAT is interfaced with a Configuration Management System in order to retrieve a complete baseline in a consistent way. After the testing has been completed, the EAT's export capability allows the user to store all test items under Configuration Control.

documentation
all testing items can be collected and formatted in an homogeneous way to produce a final Test Specification Document or Test Report Document. The contents of the documents can be tailored according to a given template and the output format can be chosen between pure ASCII or specific word processor format.

4.2.2 Process Management

One of the most interesting features of the EAT is the support it provides to manage the testing process. Although the testing life-cycle is quite similar to the software development life-cycle and has a comparable complexity, it suffers from a lack of formalisation, which makes it particularly critical. The knowledge about the testing process is formalized in a set of rules that EAT can use to drive the user to correctly perform each phases of testing both performing a consistency check and suggesting the user which activities are to be undertaken. This formalization has to take into account several factors, such as adopted standards, organisation or project constraints, common practices, and so on.

The objective of capturing this knowledge to provide the testing team with valuable assistance in a non rigid way was a major driver of the Demonstrator. A model of the testing process has been defined in terms of activities and objects that are produced by these activities in order to provide support for

- *guidance* - the model is used to provide assistance and guidance to the tester. This means that the information represented in the model is used to present the tester the actions that can be performed; and
- *understanding* - the model expresses knowledge on how to perform the testing activities. Understanding this knowledge as early as possible may help the tester to avoid the execution of erroneous activities.

As far as process management is concerned, EAT provides two kinds of support:

passive assistance
the system implements a number of consistency checks and suggests feasible or convenient activities to be performed upon user request, but it does not constrain or drive the user in the way he / she performs such activities.

active assistance
the system both monitors the user during all activities and drives him / her throughout the process. This guidance is not mandatory, but just provides the right directions to the user. This is of particular importance when testing large and complex systems, where the amount of information to be generated and managed is fairly relevant.

Different ways of formalising and managing this knowledge might be used, which are not based on knowledge representation or rule-based paradigms such as that adopted in the EAT. Nevertheless the advantage of using a proper knowledge base is that it can be easily enhanced or adapted to specific project or organisation needs. Starting from the basic knowledge on how testing must be performed (i.e. implementing the standard testing guide-lines), it is possible to add knowledge that is more specific of the application domain or of the organisation where the testing process is performed, thus capturing testers' expertise and capitalising on it.

This is particularly important for test specification and implementation, where the expertise concerning the application area as well as the peculiarities of the programming language can add a major value to these activities. No other technique would allow such knowledge to be augmented in a stepwise manner, thus building a company-specific "testing experience"

4.2.3 Further improvements

Several promising directions for further EAT evolution have been identified and are briefly discussed in the following.

Integration with external testing tools
While EAT is mainly concerned with supporting test management, the possibility to integrate external tools such as static analysers, coverage analysers or test implementation harnesses, could be useful and would make EAT a more complete tool.

Integration with other CASE tools
Some information managed by EAT is produced by other CASE tools, such as requirements analysis or design tools. It would be useful to exploit integration with these tools, in order to access such information in the right format. For instance, a CORE requirement referred by a test specification should be displayed by invoking the CORE tool directly from EAT browsers, or the whole list of requirements should be easily imported inside EAT together with the relationships between requirements, design and units code.

Fine grain support
As mentioned before, the EAT knowledge base could be enriched both by the knowledge on the nature of the software under test (functionality and structure) and by the expertise about how to test such kind of software. This implies the formalization of the previous experience on testing and the classification of the software on the basis of some paradigms, with a level of detail sufficient to correlate the experience on test definition with each class of software. With this finer level of knowledge, the user could be supported in a more effective way since the early phases of testing, and previous expertise which is normally lost after the end of a project, could be made available.

4.3 Formal Derivation of Test Cases

A large percentage of the testing effort goes into the test specification phase which is of upmost importance for its impact on the quality of the testing

process itself. In fact, functional testing concerns checking a unit or sub-system or full system against a given set of specifications.

Increasing interest in formal specification techniques for safety critical systems and real time systems in general, shows how the need for rigourous requirements formalisation is particularly felt. It has been therefore decided to investigate the feasibility and cost-effectiveness of deriving test specification from such formal requirements in a precise and possibly automatable way.

There is a widespread assumption that functional testing is not required when using formal specification techniques, because, if executable code is directly and unambiguously derived from such specification, the assessment of correctness can be performed by means of formal verification techniques.

The approach taken by Alenia is a more pragmatic one: in fact there is always a number of non-functional requirements (e.g. performance, architectural constraints, dependability) that cannot be expressed in the formal specification but strongly influence both design and implementation of an application. Therefore, even if a full automatic code generation were available, modification of the generated code could be required anyway, thus implying the need for testing.

The formal specification method chosen for this experimentation was LOTOS (Language Of Temporal Ordering Specification), an ISO standard for communication protocol definition and conformance testing, currently being investigated also by other AIMS partners companies for requirements and design specifications.

4.3.1 Alenia case study

The Demonstrator started from the formal approach to systematic test derivation defined by the CO-OP method (the name comes from the identification of COmpulsory and OPtional tests) which is applicable to a subset of the LOTOS language, the so called basic LOTOS.

The CO-OP method handles neither data specification nor variable and value declaration, but is intended to provide only a means to generate test cases from the behaviour specification of a software system. The application of the CO-OP method to basic LOTOS specifications allows to derive test cases by means of merely syntactic manipulations before they can be run against the implementation of the software system so as to verify that it complies with the original requirements specification.

The method, that is supported by a prototype tool available from the LOTOSPHERE ESPRIT project, has been enriched by Alenia AIMS team to manage data as well.

At the same time, a requirements subset of an avionic sub-system from a real project currently undertaken inside the company has been formally specified using the formal description technique LOTOS.

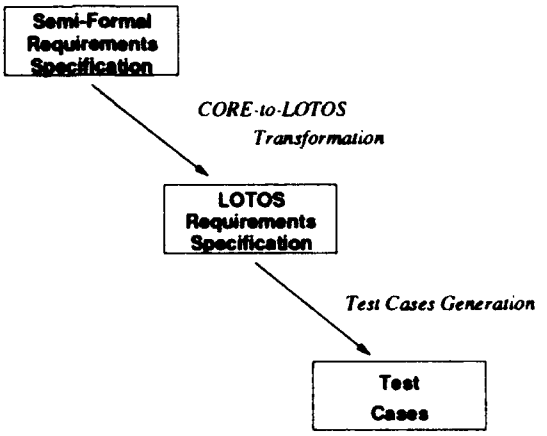


Fig. 4: Avionic system case study

The starting point of this phase was a set of requirements expressed in a semi-formal notation (CORE - Controlled Requirements Expression) and the main task was that of defining these requirements using the LOTOS notation.

The case study development activities also yielded a few general guide-lines to transform a given CORE requirements specification into a corresponding, semantically equivalent LOTOS specification. These guide-lines define a mapping from basic CORE entities into LOTOS basic entities as well as a mapping between the essential composition constructs of the two notations. Thus, for instance, the key CORE entity, the so called *action*, is mapped into the key LOTOS entity called *process*, and a sequence of actions is mapped into a sequential composition of processes. Moreover, the CORE mutual exclusion composition construct may be represented in LOTOS by means of the choice operator.

The transformation from CORE to LOTOS is not generally represented by a one-to-one mapping but shows some critical aspects, especially as far as data handling is concerned. In particular, LOTOS variables may not be used on the left side of an assignment. This implies that values are lost unless some temporary data storage mechanism is defined. This particular feature was realized in the case study by means of the introduction of auxiliary LOTOS processes whose function is merely that of acquiring a data value and returning it when requested.

The availability of a real size software system LOTOS specification represents the first step toward the evaluation of the potential of formal techniques for rigourous test specification.

The second phase, currently under way, consists of deriving the test cases in a systematic and possibly automatic way following the guide-lines of the "extended" CO-OP method.

4.3.2 Test selection

The previously mentioned test generation procedure leads to the derivation of a large number of test cases. All these test cases can detect errors in implementations, and errors detected with these test cases indeed indicate that an implementation does not conform to its specification. However, the number of test cases that can be generated may be very large, or even infinite. This implies that the execution of all generated test cases is impossible, if their number is infinite, unfeasible, if their number is very large, or simply too expensive.

The reduction of the number of generated test cases to an amount that can be handled economically and practically is therefore necessary. Such a reduction of the size of generated test suites (i.e. the reduction of the number of test cases) by choosing an appropriate subset is called *test selection by test cases reduction*. Different selection criteria may be defined in order to reduce the number of test cases once they have been generated. Nonetheless, their definition requires thorough study and understanding of the application domain and it may even vary from one system to another.

Yet another approach to test selection may be chosen which is based on syntactical manipulations of the original specification. This approach is referred to as *test selection by specification selection*. Instead of generating too many test cases and then trying to reduce their number *a posteriori*, this approach is intended to prune the generation itself.

A further approach to test selection was investigated based on the selection of critical requirements and a generation of test cases for these requirements only. This approach is called *test selection based on requirements criticality* and, up to now, it seems the most sensible in the framework of the case study.

Definite assessment of the analysed selection criteria is still under way, nevertheless a few indications may be given about the initial results. A full experiment with the approach based on test cases reduction does not seem to be feasible. In fact no currently available tool supports the automatic generation of test cases for full LOTOS specifications and the number of tests generated applying the "extended" CO-OP method is often too high to be managed by hand.

The approach based on syntactical manipulations of the original specification seems to be more promising even if it is not yet mature enough and a final statement about its effectiveness cannot be made.

Finally, the test selection approach based on requirements criticality seems to be feasible as well as sensible. It is far more pragmatic than the approaches described above and results in the reduction of the coverage provided by the generated test suite.

5 CONCLUSIONS

The testing process and its activities have been particularly neglected by R&D projects in information technology, which are usually focused on requirements capturing or design phases in software development.

Unlike many research projects, where technologies are developed for their own sake and are experimented on small toy case studies, AIMS decided to start from the real world requirements and to address them with a fairly pragmatic approach. This is also reflected by the strong commitment to technology transfer, that is already happening for the first area of the Alenia Demonstrator (testing tools exploitation) which is being transferred into the context of a real project.

Moreover each area has a different impact on the testing process. For instance, while both the use of formal techniques to derive test cases and the adoption of testing tools are fairly intrusive with respect to the common practice of development teams, EAT is orthogonal to the process, in the sense that it can be used to support different phases of the process, without imposing particular constraints on either practices or tools.

Nevertheless the effort spent in investigating the more advanced topics - that might provide usable results in the longer run - is also intended to spread the knowledge about formal specification methods that are likely to become necessary, if not mandatory, for future large programmes.

In conclusion, the AIMS experience turned to be particularly important for Alenia, since intermediate results were already transferred to operational divisions for everyday use and an improved awareness of both problems and potential solutions is being disseminated across the whole corporation. We are convinced that the main reason for this success was the very early user orientation of the project, which did not take the usual "technology push" approach, but spent a considerable amount of time and effort in understanding the most important problems developers are faced with.

The relevance of the Alenia demonstrator is such that further exploitation in various context will be considered, starting from use in different companies of the same corporation.

Discussion

Question J. BART

How many rules did you have in EAT rule based system?

Reply

≈ 100. However, this number will be increased according to the prototype evolution and the enrichment of its knowledge base.

Question C. KRUEGER

What quantitative improvement have you achieved in efficiency and product quality?

Reply

Improvements in testing efficiency and product quality will be quantified in terms of the effort spent in the different testing activities of the number of (extra) errors found in the product. Preliminary benefits have been achieved from the testing tools exploitation and their transfer to the operational teams. A reduction of 20% of the total testing effort has been achieved, even if some initial human resistance to the change has been found. Concerning the quality, some extra errors have been found in the code, especially when performing a complete white-box structural testing that has exercised parts of the code never tested before.

Validation and Test of Complex Weapon Systems

Mark M. Stephenson
U.S Air Force, Wright Laboratory, Avionics Logistics Branch
WL/AAAF-1 Bldg 635
2185 Avionics Circle
Wright-Patterson Air Force Base, Ohio 45433-7301
United States of America

1. SUMMARY

As avionics software complexity increases, traditional techniques for avionics software validation and testing become time consuming, expensive, and ultimately unworkable. New test issues arise with the development and maintenance of complex "super federated" systems like that of the B-2 and highly integrated systems like that of the F-22. Upgrades to existing weapon systems that produce a blend of federated and integrated architectures further complicate the problem. This paper discusses the limitations of current approaches, equipment and software. It defines a next generation avionics software validation and test process, along with the hardware and software components that are required to make the process work. The central goals of the process and components are to reduce development and maintenance costs, to minimize manpower requirements, to decrease the time required to perform the testing, and to insure the quality of the final product. The process is composed of unit test, component test, configuration item test, subsystem test, integration test, avionics-system test, and weapon-system test. Some of the topics covered are: automated testing of avionics software, real-time monitoring of avionics equipment, non-real-time and real-time avionics emulation, and real-time simulation. This paper is based upon several years of experience in the following areas: 1) Research and development of new technologies to improve the supportability of weapon-system software. 2) Design and implementation of facilities for the development, enhancement, and test of avionics software.

2. INTRODUCTION

Testing of avionics software has traditionally been a very manpower intensive task. For example, the manual execution of the Final Qualification Test (FQT) for the F-16A/B Expanded Fire Control Computer (XFCC) takes two engineers approximately three weeks. The F-16A/B Expanded Stores Management System (XSMS) computer software FQT requires two engineers working for eight weeks. During an FQT, the engineers sit at a simulator, read the test procedure, manually execute each step, and check for the correct results. Checking for the correct results can be as simple as checking for a symbol on a display or as complex as performing a full mathematical analysis of large amounts of logged data. In the past, this approach, although expensive, worked well and produced some very capable weapon systems. However, as the quantity and processing power of avionics computers increases, the tests tend to be proportionally more costly to develop and execute. Based upon the F-16 example, a super federated system with over ten times the processing capability of an F-16 could easily take many man-months to test. And, if software errors are found, as they often are, the software must be fixed and the tests run again from the beginning. It is very easy to see how quickly the current approach could become unworkable and lead to either major schedule slips or poorly tested software. Highly integrated advanced avionics results in even more test requirements caused by massive increases in software complexity. Features usually associated with integrated avionics like system reconfiguration and fault tolerance are very hard to test because of the number of possible configurations. The only solution is to provide engineers with tools that enable drastic improvements in productivity throughout the test process. For example, engineers that currently spend much of their time executing tests should be able to concentrate their efforts on fixing software problems and creating quality tests. The tools should take care of mundane tasks, such as stepping through a test or producing a test report. Figure 1 illustrates current avionics testing.

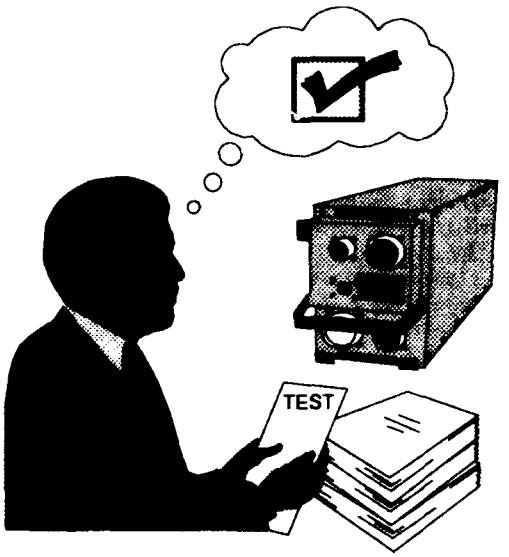


Figure 1, Avionics Software Testing

This paper proposes a set of next generation tools to support a test process for newer complex weapon-system software. Although the paper emphasizes testing, it in no way intends to diminish the importance of the requirements, design, and code generation phases of software development, or that of

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

other essential functions such as configuration management. It is a widely accepted fact that high quality development during earlier software phases decreases the number of errors, enhances the ease and quality of testing, and potentially decreases the number of tests required. In addition, the approaches suggested in this paper would fail without a comprehensive configuration management capability. The test process presented in this paper is based generally on the Department of Defense Standard 2167A (DOD-STD-2167A). This paper identifies key hardware and software tools required to support that process. The process does not assume a specific development model (i.e., waterfall, spiral); instead, each test type is identified in a logical sequence from unit test through weapon-system test. Also, the test types apply to both newly developed software and modifications to existing software. The test types, illustrated in Figure 2, are defined as unit test, component test, configuration item test, subsystem test, integration test, avionics-system test, and weapon-system test.

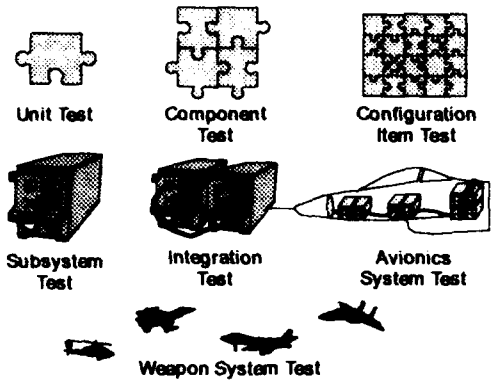


Figure 2, Test Types

3. UNIT TEST

Unit test will be the first step in testing actual code. This paper focuses on source code testing, but a full modern process might include the testing of the requirements, the design, or a system prototype model. Unit test will be done in conjunction with code generation, using an engineering workstation as shown in Figure 3. The code generation process will begin after the detailed software design is completed. The Operational Flight Program (OFP) engineers will use the detailed design to produce each Computer Software Unit (CSU) and then test each unit against remaining old requirements and the new requirements. A CSU is a nontrivial, independently testable piece of code. The CSU tests will execute with automated control and verification and will run on the engineer's workstation without the need for the target avionics computer. Code analysis tools will perform an automated CSU code walk-through. Necessary revisions will be made to the code and design documentation, followed by any necessary retesting. The development of the automated test procedures for each Computer Software Component (CSC) test will then begin.

Unit testing requires several tools; many of which support later stages of testing. The tools will be capable of software module test generation, static code analysis, automated software module testing, and non-real-time avionics computer emulation. Figure 3 illustrates unit test.

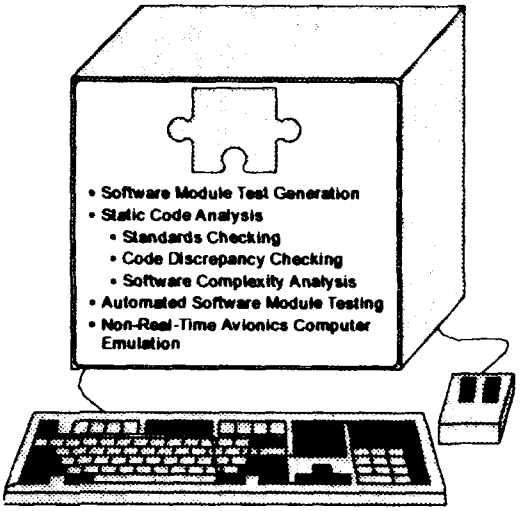


Figure 3, Unit Test

3.1 Software Module Test Generation

The goals of software module test generation are to produce a test that validates compliance with all the requirements allocated to the module, and to insure traceability between the requirements, the design, the software module, and the specific tests. The word "module" applies to a CSU, a CSC, or a Computer Software Configuration Item (CSCI). The tool will list the module requirements and provide assistance in creating the tests. Creating the tests will involve creating tables of input data and corresponding expected output data. Entering and calculating the data required for module testing can be very time consuming and error prone. Consequently, the interface to the software module test generation will be similar to a spreadsheet where data can be entered in large quantities, based on pattern sequences and formulas.

3.2 Static Code Analysis

Static code analysis has not been traditionally a part of the OFP test process because appropriate tools have not been available, especially for programming languages used in OFPs such as JOVIAL. Software technological advances and more recent high order languages like Ada have brought about more static analysis tools. Static test tools can perform such functions as identifying coding errors, like code that cannot be executed (i.e., dead code), insuring conformance to coding standards, identifying complex portions of code that may require special test attention, and suggesting a design change to improve code quality and supportability. A few static analysis tools capable of test support are listed below. Increased static code analysis capability is expected in the future. Static code analysis will include standards checking, code discrepancy checking, and software complexity analysis.

3.2.1 *Standards Checking*

Standards checking will validate the compliance of the developed source code with the project programming standards and guidelines. Standards checking will include a review of naming conventions, module size, frequency of comments, readability, etc. A detailed report will list any deviations from the standards.

3.2.2 *Code Discrepancy Checking*

Code discrepancy checking will identify potential discrepancies in the code; i.e., code that cannot be executed, variables that are declared but never used, and control flow that has a single path (i.e., if 1<2 then ...). In addition to identifying quality and efficiency problems in the code, discrepancy checking will identify potential errors by drawing attention to portions of the code that may contain errors.

3.2.3 *Software Complexity Analysis*

Software complexity analysis will evaluate source code for complexity. The tools will analyze the source code using complexity metrics such as that of McCabe and Halstad. The tools will identify portions of code that are potentially too complex for complete testing and cost-effective maintenance.

3.3 Automated Software Module Testing

Unit testing has always been labor intensive. The basic problem is the fact that units are usually embedded in a larger body of software. The tasks of separating out each unit, writing the required test driver software for each CSU, and executing the test are very manpower intensive. In addition, a lot of software is produced that will eventually be discarded. An automated software module test tool will extract units from the existing software based on comments in the code, generate the test shell, execute the test, and report the results. The report will include general information on the unit that was tested, the test that was executed, the results of the test, and the test coverage analysis. The coverage analysis will help the engineer to understand how comprehensive a test is by identifying the number of times each line of code is exercised during the test. As software development continues, the tool will maintain a log of tested units and input test data. The logged information will then be used to indicate which unit tests need to be repeated due to changes in the unit code or changes in the unit test data. An OFP developer will make modifications to the software and then invoke an automated unit test. The automated unit test tool will then identify all units that need to be tested and execute the tests. New units without test data will be identified to the user.

3.4 Non-Real-Time Avionics Computer Emulation

A non-real-time emulator hosted on the engineer's workstation will support the execution of automated tests. The emulator will provide an instruction set simulator for the target processor, simulation of all the hardware that the software communicates with, and a full interactive debugging capability. The debugger will provide visibility into the execution and data values of the software module under test. It will support assembly language or high order language "views" of the software. The emulator will also produce a variety of calculated estimates of the execution characteristics of the software running on the actual avionics computer, including memory requirements and execution timing.

4. COMPUTER SOFTWARE COMPONENT TESTING

The Computer Software Component (CSC) testing will begin after the coding and CSU testing have been completed. The CSC testing will insure that the algorithms and logic are correct and that the CSC satisfies its allocated requirements. DOD-STD-2167A's CSUs, CSCs, and CSCIs imply that software can be broken into three logical levels. In practice, the number of component levels should reflect the complexity of the software. Tests should be executed according to the requirements of each level. Therefore, CSC testing will include testing groups of CSCs in addition to groups of CSUs forming a CSC. All the CSC testing will be performed on workstations using non-real-time avionics computer emulators and automated software module testing. Corrections to the code will result in revisions to the documentation, retesting, and updates to the configured CSU and CSC source files. A code analysis, assisted by automated tools, will be performed on any code that has been revised. The CSCI automated software module tests will then be developed and checked for completeness and accuracy.

Due to the strong similarity between CSU and CSC testing, CSC testing will use all of the tools used in CSU testing, including software module test generation, static code analysis, automated software module testing, and non-real-time avionics computer emulation. However, CSC testing will place additional demands on the tools identified in section 3. The new requirements include support for multiple levels of testing and support for software physically located in multiple files. No new tools will be required to support CSC tests. Figure 4 illustrates CSC testing.

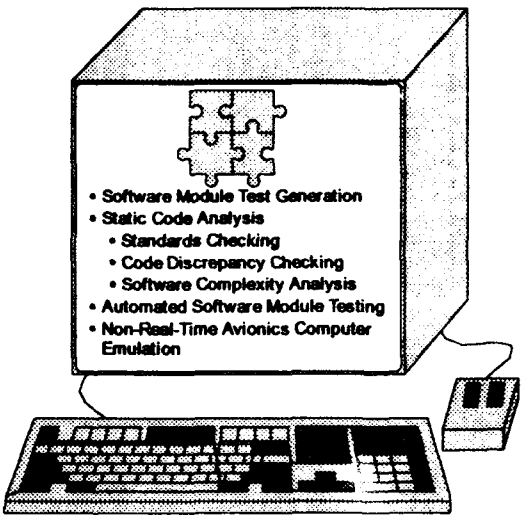


Figure 4, Component Test

5. COMPUTER SOFTWARE CONFIGURATION ITEM TESTING

The Computer Software Configuration Item (CSCI) testing will begin when CSC testing is complete. The Operational Flight Program (OFP) engineer will produce a full CSCI load

module and execute it on the workstation, using a non-real-time avionics computer emulator. The goals of CSCI testing will be to insure that all the components function together, and to verify that the interface to other CSCIs is in accordance with the interface design document. To reduce equipment costs, the test will be conducted on a workstation rather than on the more expensive avionics computer. Depending upon the relative processing capability of the actual avionics computer and the workstation executing the emulation, the utility of this stage of testing may vary. If the execution of the workstation emulation is excessively time consuming, then this testing can be done during subsystem test. The non-real-time avionics computer emulator will have full CSCI debug capability that will include monitor and control of the CSCI execution and its data. Automated module testing software will provide test data to the CSCI and then verify that the corresponding CSCI output data is correct. The tests will be performed and the results recorded in the software test report. Revisions to documentation or code and retesting of the CSUs, CSCs, and CSCI will be directed by the results of this workstation-based testing.

The CSCI testing will use many tools used earlier in CSU and CSC testing. These tools include software module test generation, static code analysis, automated software module testing, and non-real-time avionics computer emulation. No new tools will be required to support CSCI tests. Figure 5 illustrates configuration item testing.

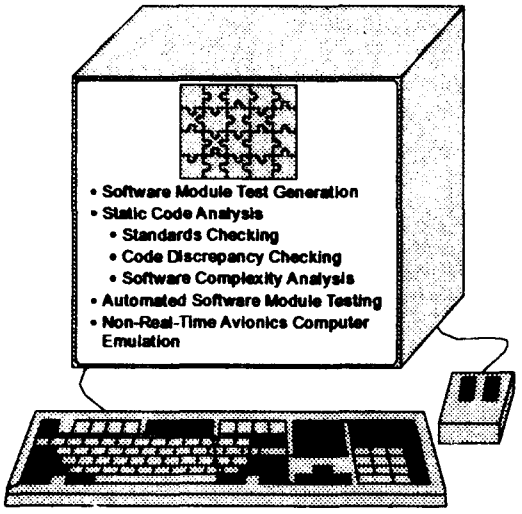


Figure 5, Configuration Item Test

6. SUBSYSTEM TESTING

Subsystem testing begins after the workstation-based CSCI testing phase has been completed. During subsystem testing, the OFP engineer will use a test station to load the actual OFP(s) into the avionics subsystem hardware. This will be the first time that the new software will be executed on the actual avionics hardware. Therefore, these tests will focus on software integration with the hardware and include tests of external communication with other subsystems. Subsystem testing will be performed with both statically and dynamically generated data.

Internal interaction and basic external interface testing will be initially performed using statically generated data. The subsystem will be stimulated with specific inputs and the outputs will be verified. Inputs to the subsystem will be produced from pre-generated data, and then during execution, the inputs will be driven by the real-time requirements of the subsystem. The outputs will be monitored in real time and verified for correctness. Depending upon the particular subsystem, the inputs and outputs can take a variety of forms (e.g., discrete data, serial digital data, parallel digital data, analog data, and Radio Frequency (RF) data). Complete computer control of the interfaces/devices that produce and monitor the inputs and outputs will be provided. In addition, the OFP engineer will have full real-time monitor and control capabilities over the subsystem computer(s). With these capabilities, the OFP engineer will be able to initiate stimulation of the subsystem, monitor the response, and observe the execution of the software. Thus the OFP engineer will be able to verify conformance to interface requirements, check the basic operation of the software, and isolate any discrepancies in the software.

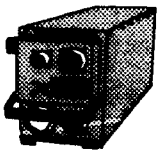
The second level of testing will be fully dynamic and scenario oriented. The environment external to the weapon system will be simulated, and the various avionics that communicate with or produce data for the subsystem will also be simulated. The simulated environment together with the simulated avionics will provide a fully interactive, real-time environment by supplying data to and from the real-time subsystem interface. The subsystem will be "flown" through various scenarios in a realistic controlled environment that will exercise the subsystem software in a variety of dynamic conditions. The tests will be specifically designed for the subsystem under test. The test station simulation and real-time avionics monitor and control capability will be used either manually or automatically to execute tests, to verify correct operation, and to isolate any discrepancies in the software. The relative magnitude of the tests done using statically generated data verses those done using dynamically generated data will depend upon the operational function of the subsystem under test. Subsystems with more dynamic functions such as weapon delivery or target tracking will require more dynamic testing. Subsystems like weapon stores management and displays will be likely to require more tests using statically generated data. After both subsystem test types have been accomplished, the results will be recorded in a software test report. Any revisions to documentation or code and retesting of the CSUs, CSCs, and CSCIs will be based on the results of the subsystem level testing.

It is important to note that there will be (and there currently are) two distinct types of OFP engineers executing the tests: the OFP developer and the OFP tester. Each type will have a different view of testing and different requirements for test support tools. The OFP developer will write the software and will conduct initial tests. This function will require extensive monitoring of both the execution of the OFP undergoing testing and the test-station data stimulating the OFP. While

the OFP is executing in the avionics computer, the OFP developer will need extensive interactive debugging capabilities at both the assembly language level and in the high order language symbols and statements. The OFP developer will focus on the software's behavior as it executes internally in the avionics computer. This approach is sometimes called "white box testing." The tests will range from unit to avionics-system tests. Some testing will be automated; however, many tests will be highly interactive as new functions are checked and problems are found and isolated.

OFP testers will usually be managerially independent of the OFP developers and will have a different test focus. The OFP tester will treat the avionics computer and software like a "black box" that will be expected to provide certain functions. The OFP tester will focus on exercising the avionics computer in many different modes and scenarios to insure that it will function according to its requirements. A primary interest of the OFP tester will be the automation of the long and monotonous formal tests. Automating the tests will also permit the OFP tester to design and perform more comprehensive tests. The OFP tester will perform manual and automated, subsystem, integration, and avionics-system testing.

During subsystem testing, OFP developers and testers will use numerous hardware and software tools. These tools will also support integration testing, and some will support avionics-system testing and weapon-system testing, as well. These tools include a Virtual Test Station (VTS), OFP test data generation, OFP test data driver, test case generation, automated real-time testing, data monitor and control, and real-time avionics computer emulation. Figure 6 illustrates subsystem test.



- Virtual Test Station
- OFP Test Data Generation
- OFP Test Data Driver
- Test Case Generation
- Automated Real-Time Testing
- Data Monitor and Control
  - Simulation and Stimulation Monitor and Control
  - Avionics Communication and Interface Monitor and Control
  - Avionics Computer Monitor and Control
- Real-Time Avionics Computer Emulation

Figure 6, Subsystem Test

6.1 Virtual Test Station

When aircraft had only one or two embedded computers, avionics software was traditionally supported by an avionics test station (sometimes called a "dynamic test stand" or an "integration support station"). The test station simulated all of the necessary subsystems in real time, provided stimulus signals to an avionics computer running the software under test, and monitored and recorded outputs produced by the software. The test station, which was custom built for each type of avionics computer, had a large simulation computer that performed the real-time computation. Complex, special-purpose interfaces provided stimulus and monitoring signals for the avionics computer. These systems were very expensive to develop, and were also found to be very expensive to enhance and maintain. For aircraft with only a few avionics computers, this approach was viable, but inefficient. For modern aircraft with many avionics computers, large, monolithic, dedicated test stations are no longer practical. The future requires us to break this cycle of building large, dedicated test stations for each avionics computer, and to develop a collection of modular, generic components from which test stations can be assembled easily and economically. By emphasizing flexibility, extendibility and reusability in the design of the test station architecture and components, the same set of components will be able to support software on multiple avionics computers.

In addition to lowering cost and increasing flexibility, other avionics support requirements are emerging due to super federated and integrated avionics systems. As avionics systems become more integrated, the support systems that will be used to test the avionics software and hardware will need to be more integrated. Early in the development of avionics software, the work will focus on individual avionics subsystems in isolation. As development progresses, more of the actual avionics subsystems will be incrementally integrated until the entire avionics suite is complete. The engineers will need a test station that allows them to work on individual subsystems independently without interfering with others. Then, when they are ready to integrate, they will need a test system that can be easily reconfigured to incorporate any number of the other subsystems required for a test. The VTS will address these issues.

The VTS will be a completely reconfigurable avionics support system. The reconfiguration will be controlled through software, thus permitting rapid, easy changes to the test system. It will reconfigure to support a variety of tests using various combinations of avionics hardware and software. Working from either a workstation or a test console, an OFP engineer will use the VTS to configure a test station with the required avionics hardware and simulation software to support a particular test. With the VTS, permanent individual test stations will not exist. The support environment will consist of a collection of hardware and software building blocks that can be configured to produce exactly the test station configuration required for any given test. The VTS will contain workstations, test consoles, avionics computers, avionics computer interfaces, avionics computer stimulation equipment, avionics computer monitors and controllers, and general purpose real-time processors. The VTS will include a combination of models, avionics emulators, and actual avionics hardware, which will provide the environment for the avionics subsystem under test and exercise it in various scenarios. The VTS will be capable of performing tests with one actual avionics subsystem and the other avionics and external environment simulated. It will also perform integration testing (discussed in section 7) with multiple actual avionics subsystems, while simulating the remaining avionics and the external environment. Assuming there are sufficient resources within the VTS, it will support many test stations in various configurations, all operating simultaneously. When a specific component of the VTS is not in use, it will be available from the pool of VTS resources to be allocated and used as part of a test station. This will allow maximum use

of the costly avionics and simulation resources. In addition, OFP engineers will have access to the hardware and software needed to perform any test. Figure 7 illustrates the VTS resources.

The VTS concept provides many interesting possibilities. From the perspective of the OFP engineers, it will be possible to select exactly the resources or resource types needed for a particular test. From a test-station maintenance perspective, it will be possible to take various resources "off-line" for maintenance without impacting the use of the other resources. If a resource fails while allocated to a particular user, the user will simply take that resource off-line, replace it with an equivalent resource, and then alert the maintenance organization. From the test-station management perspective, the VTS concept will significantly reduce the cost of building and maintaining a full avionics software support facility. Since test station resources will not be permanently configured in a VTS facility, more efficient use of a smaller aggregate number of resources will provide the same effective testing capacity as a larger, more expensive, traditional facility. This will reduce initial acquisition costs and life-cycle costs. Also, to optimize use of VTS resources, the VTS resource allocation will be automatically monitored to determine when more resources are required to support the work load, and when existing resources are not utilized and can therefore be eliminated.

For more information on the VTS, reference "Virtual Test Station" by Steven A. Walters of Science Applications International Corporation (SAIC) and Mark M. Stephenson of the United States Air Force, a paper presented at the National Aerospace and Electronics Conference (NAECON), May 24-28, 1993.

6.2 OFP Test Data Generation

The goals of OFP test data generation are similar to those of software module test generation; i.e., to produce a test that validates compliance with the requirements allocated to the subsystem software, and to insure traceability between the requirements, the design, the subsystem software, and the specific tests. Test data will be generated for requirements that can be adequately and practically validated with statically generated data. The test generation will focus on defining the stimulation data with its associated timing requirements, along with expected response data and timing. The specific stimulation and response data types will depend upon the subsystem under test. Many subsystems now communicate using the Military Standard 1553B (MIL-STD-1553B) bus, analog signals, and discrete signals. Other subsystems may require and/or generate RF or optical signals. Entering the raw data to control and monitor these signals can be very time consuming and error prone. Consequently, the interface for test data generation will be similar to a spreadsheet where data can be entered in large quantities, based on pattern sequences and formulas. Timing will be defined using various options; e.g., a function of time, a function of the avionics computer's major processing cycle, and a function of the demand of the avionics and stimulation hardware.

6.3 OFP Test Data Driver

The test data driver will be similar to the automated software module test tool but will have some important differences. The test data driver will control the VTS and drive a sequence of real-time data to produce actual input signals to the avionics. In addition, the test data driver will check the real-time output. For example, the MIL-STD-1553B bus will be driven with realistic data and the MIL-STD-1553B response will be checked for correctness. Timing of the input data and the responses is critical, and in fact, the measurement of the

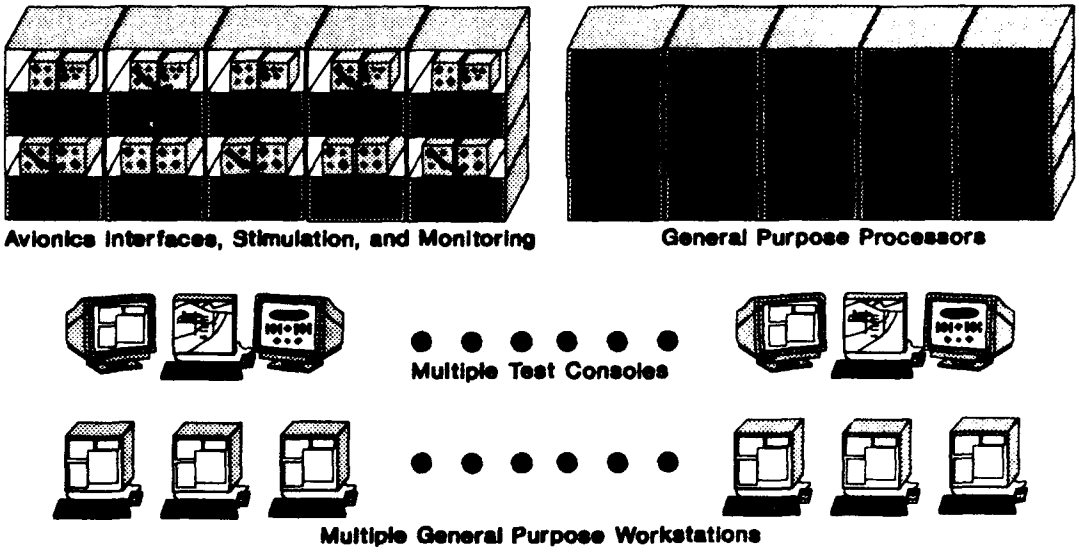


Figure 7, Virtual Test Station (VTS) Resources

time between events will be one of the test options. A report will be generated by the OFP test data driver that will include general information on the OFP that was tested, the particular test that was executed, and the results of the test.

6.4 Test Case Generation

The goals of the test case generation, like those of the software test data generation, are: to produce a test that validates compliance with the requirements allocated to the subsystem software, and to insure traceability between the requirements, the design, the subsystem software, and the specific tests. Test case generation will be used to create automated test procedures that can be adequately and practically validated with dynamic scenario-oriented testing. Test cases will be written in a procedure-oriented, structured, english-like language that can be easily understood by a test engineer. The test case generation tool will be "language sensitive" with features like automatic syntax checking and mouse-driven selection of language constructs and key words. The tool will have a graphical representation mode to provide the user with a "picture" of the total test flow. Since the complexity of the software is increasing drastically, the generation of quality tests is becoming increasingly complex. The test case generator will provide a variety of techniques to support automatic generation of test sequences.

The test case language will support the full representation and automation of the actual test and validation procedures. The test language will also have provisions for the execution of special debug tests to isolate problems found by the test. The test language will include a variety of constructs including: 1) command file structuring; i.e., including other command files and defining subroutines, 2) simulator set up and control; i.e., selecting the simulation configuration, loading initial conditions and stopping the simulation, 3) user control; i.e., turning a cockpit panel knob, or flying the aircraft, and 4) validation and analysis; i.e., checking that a cockpit light comes on or that a MIL-STD-1553 packet contains certain data, or checking a set of logged data for a given mathematical relationship.

6.5 Automated Real-Time Testing

The automated real-time test software will "fly" the simulator through the test scenario and perform the appropriate tests. If discrepancies are found, appropriate actions will be taken and the problem will be logged to a test report file. Since the test command language will permit the automatic selection and configuration of VTS resources, tests scheduled for execution after hours will be able to automatically allocate the required VTS resources, execute, and then deallocate the resources. According to estimates based on prototyping portions of actual tests, automated real-time testing will effectively increase the total test case execution speed by over 100 times. Figure 8 illustrates automated real-time testing.

In addition, a potentially very valuable test technique will be made possible by the capability of fully automated real-time testing. As avionics software becomes more sophisticated, the number of paths through the software drastically increases. It would take many years to go through every possible software combination on even a current generation weapon system. Instead, tests today are developed to test major functions in

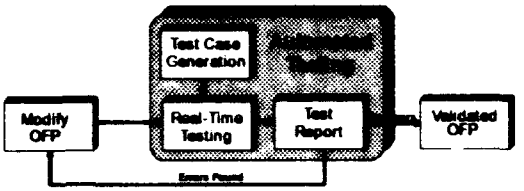


Figure 8, Automated Real-Time Testing

the software. Only extremely critical functions such as nuclear weapon delivery have very extensive tests. However, if an automated real-time test capability can operate totally without user intervention and simulators can be produced at minimum cost, then testing could occur in two phases. The first phase would include the basic critical tests that are done now. The avionics software could then be released. The second phase would be an exhaustive, random test that would run continually (or whenever the simulation resources are idle) for the life of the software version. The goal of this test would be to find any unique combination of circumstances that exposes a software problem. If a major problem were found, an emergency correction could be made. If the problem were minor then the correction could be incorporated into the next version of the software.

6.6 Data Monitor and Control

Real-time data monitor and control is a critical function in both debugging and validating avionics software. The OFP engineers need to know exactly what information is going in and out of the subsystem under test, and they need to monitor the execution of the software in order to either find software problems or demonstrate its correctness. The OFP engineers also need the capability to modify data in real-time to create unique situations; i.e., corrupt communication data, and failures. Data monitoring and control can be extremely complex. Test stations and avionics produce large quantities of data at very high rates, making storage of all the data impractical. Also, the data that is produced in multiple locations (e.g., avionics subsystem, simulation) must be synchronously controlled and monitored, and then correlated, in order to provide information on the entire system. A common data time tag will be used to merge and analyze system-wide data. Finally, it is generally unwise to stop or delay the simulation and avionics to obtain data. Real-time systems (especially avionics) often change state when they are halted, and they cannot always be restarted at the point from which they were halted.

Data monitoring will include multiple alphanumeric and graphical color display types to present the information in a format that can be quickly interpreted. A common user interface for both monitor and control will be independent of the source of the data and of any special hardware and software that might be required to modify or obtain the data. Also, multiple filtering and data event triggering functions will reduce the time and effort required to understand the data and to compare it against expected results. Three primary functions are required for data monitoring and control: 1) simulation/stimulation monitor and control, 2) avionics communication and interface monitor and control, and 3)

avionics processing monitor and control. Figure 9 illustrates the three types of monitor and control.

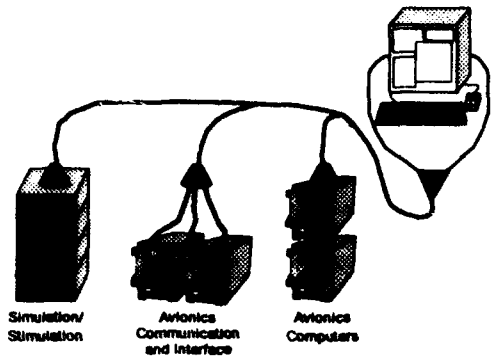


Figure 9, Monitor and Control Types

6.6.1 *Simulation/Stimulation Monitor and Control*
In an automated or manual test, the test engineer requires full access to the simulation and avionics stimulation data to know, in detail, the environment provided for the subsystem under test. The test engineer also needs to be able to control the environment to generate various tests. The simulation/stimulation monitor and control will allow the manual or automatic user-specified monitoring, logging, and controlling of simulation and stimulation functions and data.

6.6.2 *Avionics Communication and Interface Monitor and Control*
Most avionics computers have multiple digital and analog interface signals to send or receive information from other avionics computers, devices, sensors or indicators. Most avionics computers also have a general purpose communication bus (e.g., MIL-STD-1553, High Speed Data Bus) for data transfer among the other avionics computers in the weapon system. Newer weapon systems usually have many communication busses. Occasionally, avionics computers also have a dedicated high speed communication channel for transfer of large amounts of data (usually minimally processed data) directly from one point to another. In general, for complete testing, all the avionics communications and interfaces need to be monitorable and controllable. Monitoring is complicated by the higher data rates of the communication busses and especially by the dedicated high speed channels. Monitoring these signals usually requires special purpose hardware. The ability to filter unneeded data and trigger on events will reduce the quantity of the data and extract the precise information that is required for the test. Control has some technical limitations. For example, a dedicated high speed channel that connects two avionics boxes in the test station may have some very strict timing and handshaking requirements. A specially-designed hardware device inserted between the two avionics boxes may not be able to receive, detect, alter, and transmit the communications and still meet the timing requirements. This problem can usually be overcome by simulating one of the avionics boxes and driving the high speed signals with an interface. The simulation can then be used to alter the data while driving the interface. The flexibility of the VTS will enable changes in the monitoring equipment and avionics configurations in accordance with the current test requirements.

6.6.3 *Avionics Computer Monitor and Control*
Both a manual and an automated monitor and control capability are essential for testing and debugging avionics software executing in an avionics computer. This capability will be used to load software into the avionics computer and observe the software executing in real time. Monitor and control functions will include software download and upload; memory reads, writes and searches; register reads and writes; avionics hardware halt, run and reset; instruction single step; breakpoint set and clear; timing between events; event-triggered data recording; and event-triggered program trace.

Several technical issues are associated with halting, continuing, and single-stepping avionics subsystems. First, once an avionics subsystem is halted, it typically becomes very complicated to resume real-time operation at the point where it left off. Internal timers, interrupts, and other time-critical and status-dependent hardware functions tend to change state after the halt, and it is often impossible to reliably reinitialize the hardware to enable it to continue. It is usually best to reset the hardware to resume operation. Secondly, single-stepping is an effective way to test the basic internal logic of the software, as long as the user understands the limitations of non-real-time operation and the problems with continuing real-time execution. Thirdly, single-stepping the simulation software and multiple avionics computers is especially difficult. The definition of a single step for a system with multiple processors executing at different clock rates is a basic problem. In addition, all of the processors have to be synchronously stepped. A full-featured real-time monitor and control capability is usually a better approach than single-stepping. The problems of halting and continuing the processors are avoided, and the results are more trustworthy because they are closer to actual operation.

Traditionally, real-time avionics computer monitor and control has been very expensive because of the complexity of monitoring a processor, the speed of the events that have to be monitored, and the lack of readily available interface signals that support monitoring. Also in the past, a strong effort has been made to insure that the timing of the processor is not altered by the monitoring equipment. As processors evolve, new hardware features such as cache memory, pipeline processing, superscalar processing, high speed processing, multi-processing, and "computers on a chip" have made it difficult to monitor the processor. Newer software techniques, such as dynamic variable memory allocation and dynamic software task reconfiguration, increase the complexity of tracking the execution of the software. Consequently, future avionics subsystems will require part of the actual avionics software to be dedicated to real-time testing. Furthermore, since the avionics will be tested with the embedded test software, that software must remain in the final operational software to avoid invalidating previous tests. The avionics software together with special monitoring hardware will provide a full avionics computer monitor capability.

6.7 Real-Time Avionics Computer Emulation

The real-time hardware emulator will be capable of loading an actual unmodified OFP and executing it at the same rate as that of the actual avionics computer. Real-time avionics computer emulators are usually implemented in hardware to meet throughput and other timing requirements. The real time emulator will be used for several purposes. 1) It will act as a high fidelity model to improve the simulation accuracy and to reduce the maintenance costs of the simulation. 2) It will be used to debug software for avionics computers that are very hard to monitor. Since the emulator will be a laboratory based tool designed for testing, all the available signals and "hooks" will be added to support a full avionics monitor and control capability. 3) It will save equipment costs by avoiding the need for special-purpose stimulation, such as high frequency RF signals, which might be required by the actual avionics computer it emulates. 4) It will be much less expensive to produce, acquire, and maintain than the actual avionics computer. The real-time avionics computer emulator has many advantages, however, it will never replace the actual avionics equipment in the laboratory. It is a very useful tool, especially in the early stages of testing and when facility funding is limited. Figure 10 illustrates real-time avionics computer emulation.

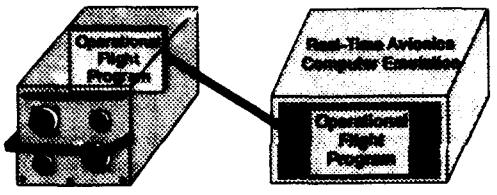


Figure 10, Real-Time Avionics Computer Emulation

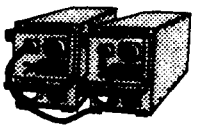
7. INTEGRATION TESTING

Integration testing will begin when subsystem testing is complete. During integration testing, the avionics subsystem will be incrementally integrated with the other avionics subsystems in the weapon system. The OFP engineer will determine how well the OFPs functions in concert with the other avionics computers and devices, and will test higher level functions that were allocated to multiple subsystems. Depending upon the level of integration and the number of avionics computers in the weapon system, integration testing will consist of one or more steps. Each step will incorporate additional avionics subsystems and will bring the test system closer to the real weapon system. The VTS will be used to produce test station configurations with various combinations of real avionics, emulated avionics, and simulated avionics, as the software progresses through integration testing.

Like subsystem testing, basic interface testing will initially be performed with statically generated data when appropriate for the subsystems configured in the test station. The subsystems will be stimulated with specific inputs, and the outputs will be verified. The inputs to the subsystems will be produced from pre-generated data and then executed according to the real-time requirements of the subsystems. The outputs will be monitored and verified for correctness. The OFP engineer will have full monitor and control capability over all the subsystem computers. With this capability, the OFP engineer will initiate stimulation of the subsystems, monitor the responses, and observe the execution of the software. Thus the OFP engineer will be able to verify conformance to interface requirements, check the basic operation of the software, and isolate any discrepancies in the software.

Like subsystem testing, the second level of testing will be fully dynamic and scenario oriented. The environment external to the weapon system will be simulated, and the avionics components not present in the given test station configuration will be simulated to provide a fully interactive, real-time environment. The subsystems will be "flown" through various scenarios in a realistic controlled environment able to exercise the subsystem OFPs in a variety of dynamic modes. The tests will be designed for the specific subsystems under test. The tests will focus on verifying integrated functions performed by multiple subsystems. The simulation and real-time avionics monitor and control capability will be used either manually or automatically to execute tests verifying correct operation, and to isolate any discrepancies in the software. Once both integration test types have been performed, their results will be recorded in a software test report. Revisions to documentation or code and retesting of the CSUs, CSCs, and CSCIs will be based on the results of the integration testing.

The integration testing will use all the tools used earlier in subsystem testing with one additional requirement. The avionics computer monitor and control will support multiple avionics computers by providing a common time tag and cross triggering of events. For example, an event in one avionics computer (e.g., subroutine call, branch taken, variable out of range) will be able to cause instruction tracing to begin on another computer. The data from both avionics computers will then be correlated and analyzed. Multiple avionics computer monitor and control will give OFP engineers a system-level view to support the testing and debugging of highly integrated functions. Figure 11 illustrates integration test.



- Virtual Test Station
- OFP Test Data Generation
- OFP Test Data Driver
- Test Case Generation
- Automated Real-Time Testing
- Data Monitor and Control
  - Simulation and Stimulation Monitor and Control
  - Avionics Communication and Interface Monitor and Control
  - Avionics Computer Monitor and Control
- Real-Time Avionics Computer Emulation

Figure 11, Integration Test

8. AVIONICS-SYSTEM TESTING

Avionics-system testing will begin once integration testing is complete. During avionics-system testing, almost all of the actual avionics will be present in the test station. The electrical characteristics (e.g., electrical noise, cross coupling

of signals, power fluctuations) of the avionics hardware within the test station will be nearly identical to that of the actual aircraft. The OFP engineer will determine how well the software functions within the full avionics system. A System Test Station (STS) will be used to perform the testing. Most of the avionics-system testing will be performed manually. Automated testing will be limited by the requirement for a large number of actual avionics hardware, and the requirement to maintain the actual electrical characteristics of the avionics system. Testing will be conducted in both stationary mode and simulated flight mode. Stationary testing is like testing an aircraft parked on the runway. It allows the maximum amount of avionics to be physically present in the system. Many tests not requiring flight will be conducted in this way. Tests that need aircraft flight require simulation of the external environment and aerodynamics. In addition, enough of the avionics will be simulated to make the remaining avionics "think it is flying." Also, sensors have to be either dynamically stimulated or removed from the configuration and simulated. Simulated flight enables testing of the dynamic functions. During avionics-system testing, additional system characterization measurements and tests will be run to measure and test functions that cannot be tested without most of the real avionics in a system. For example, timing is especially critical when dropping bombs on a target. The time that elapses between the stores management system's command to release a bomb and the actual release of the bomb must be measured accurately because of the speed of the aircraft. The software has to compensate for the delay in transmitting the bomb release command, having the bomb-release hardware receive the command, and physically releasing the bomb. This type of measurement can only be made with the actual hardware. The stationary and simulated flight avionics-system tests will be performed and the results of the tests will be recorded in a software test report. Revisions to documentation or code and retesting of the CSUs, CSCs, and CSCIs will be based on the results of the avionics-system-level testing.

Avionics-system testing will use some of the tools used in integration testing including partial use of the VTS, test case generation, automated real-time testing, data monitor and control, and real-time avionics computer emulation. The avionics-system testing will be conducted using the System Test Station (STS). Figure 12 illustrates avionics-system test.

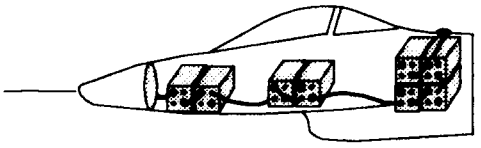


Figure 12, Avionics-System Test

8.1 System Test Station

System Test Station (STS) based tests will be the final stage of testing prior to weapon-system test. Therefore, it is critical that the avionics operate very close to the way it will operate on the aircraft. The STS has traditionally been called a "hot bench" or a "hot mock-up." It will contain a nearly complete set of avionics, a few avionics computer interfaces, a few avionics models, and environment models. The simulation resources will be allocated from VTS resources. Monitoring will be restricted only to those signals that can be shown to be unaffected by the addition of monitoring equipment. A typical weapon system will require multiple STSs to support different fielded avionics hardware configurations. Since the STS is very similar to the actual weapon system, it can also support validation of the VTS hardware and software. Figure 13 illustrates a system test station.

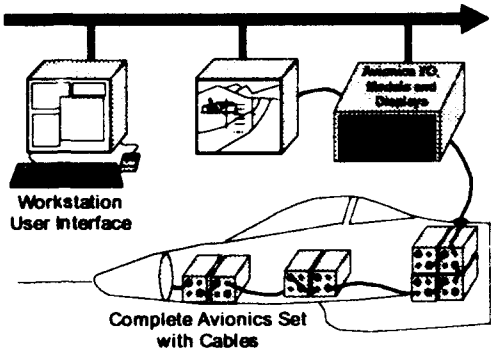


Figure 13, System Test Station

9. WEAPON-SYSTEM TESTING

Weapon-system testing will begin when all ground-based testing has been completed. Weapon-system testing will insure that the new software functions correctly with the entire weapon system, that the system-level requirements are met, and that any lower level requirements not readily testable in a ground based laboratory are tested. Throughout the test process, portions of tests may be deferred to later stages of testing because of difficulty in producing the actual test. Tests will use the most technically appropriate and cost effective approach. Weapon-system testing will involve loading an actual test aircraft with the software changes and testing it first on the ground and then in actual flight. The weapon-system tests will be performed, and the results of the tests will be recorded in a software test report. Revisions to documentation or code and retesting of the CSUs, CSCs, and CSCIs will be based on the results of the weapon-system testing. Figure 14 illustrates weapon-system testing.

Figure 14, Weapon-System Test

9.1 Flight Test Aircraft and Range Assets

Flight test will require a fully instrumented aircraft and associated test range assets. (The author lacks experience and knowledge in the specifics of flight test; therefore, specific requirements for the flight test aircraft and the test range are not provided.)

9.2 Flight Test Data Analysis

Due to the cost of flight test there is a tendency to log a lot of data. Consequently, most flight tests produce large quantities of data that are never used. Since the focus of collecting the test data is the flight test itself, flight test data is usually not used for other functions, such as validating simulation models and emulators or supporting stimulation of the avionics with actual flight data. Three main capabilities will be required for flight test data analysis. First, an automated data processing and data visualization capability will be used to reformat the test data and then perform automated data analysis or manual data visualization. Manual data visualization will use an interactive color graphics display to present the data in various alphanumeric and graphical displays, which will allow an OFP engineer to quickly understand and analyze large quantities of data. Second, the flight test data will be used to support validation of the simulation models and emulators by comparing simulation data with actual flight test data. Third, the VTS will be used to "play" real flight test data back in the simulation in order to stimulate the software with real aircraft data and to reproduce problems in the laboratory that were identified in flight test. The avionics computer monitor and control capability will then be used to monitor the processors and find the software problem. The limitations of flight test data playback are potentially significant. Due to logging capacity and instrumentation limitations, flight test data is usually incomplete. It is often hard to recreate the initial conditions of the avionics prior to turning on flight test data collection. In addition, data needed to make the playback complete is frequently not instrumented and logged during the flight test. For example, the avionics data busses are usually logged, but many discrete and analog signals may not be logged. When recreating the flight-test events in a ground based simulation, the data bus information can be played back appropriately, but the discrete and analog signals have to be synthesized in some way. Without the full set of signals, it may be nearly impossible to recreate the full flight test scenario.

10. Conclusion

This paper was written to document a baseline understanding of a next generation process and tool set required to support validation and test of complex weapon systems. The process and, especially, the tools are constantly evolving due to technological advances and continued research. The United States Air Force, Wright Laboratory, Avionics Logistics Branch (WL/AAAF) has conducted research in this area for many years and has supported numerous research projects centered on the process, methods, and tools referred to in this paper. Some of the research areas, names of the projects, and points of contact are listed in the table below. WL/AAAF recognizes the magnitude of avionics test problems and continues to team with other organizations, extending research to discover new solutions.

RESEARCH AREA	PROJECT NAME	POINT OF CONTACT
Test Technologies	Future Embedded Computer System Support Technologies	Mark M. Stephenson
Static Code Analysis	Avionics Software Design Complexity Measure	Kenneth Littlejohn
	Complexity Metrics for Avionics Software	Kenneth Littlejohn
	Avionics System Performance Measure	Clive L. Benjamin
Automated Testing	Advanced Avionics Verification & Validation	James S. Williamson
	Automated OFP Validation	Mark M. Stephenson
	Automatic Programming Technology for Test Program Set Software Generation	James S. Williamson
Test Stations	Advanced Multi-Purpose Support Environment	Capt Lloyd Ramey
	Common Modular Environment	John Luke
	Virtual Test Station	Mark M. Stephenson
Avionics Stimulation	A Digital Avionics Methodology Schema	Joe Cardenas
Data Monitoring	Multiple Avionics Processor Monitor	Lt Tony DelSorbo
	High-Speed Avionics Data Instrumentation System	Anson Dixon
Avionics Computer Emulation	Real-Time Avionics Computer Emulator	Charles Hicks
Aircraft Instrumentation	Data Integration & Collection Environment System	Tod J. Reinhart
	Operational Avionics Smart Instrumentation Systems	Larry Huntley
	Multi-Source Smart Interface Controller	Jong Hwang
Data Visualization	Avionics Data Visualization Integration Support Environment	Jong Hwang

Table 1, WL/AAAF Related Projects

Discussion

Question J. BART

Can you use pregenerated tests to drive higher levels of testing?

Reply

Yes you can, but the time and engineering required to generate the data make it very difficult. Avionics typically cycles at at least 50 times per second. Vast amounts of data are required for even a short test.The higher the level of test, the more data must be pregenerated. For this reason, both dynamically and statically generated data should be provided, but statically generated data tend to be used in the earlier stages of testing. Dynamically generated data tend to be used at later stages.

TESTING OPERATIONAL FLIGHT PROGRAMS (OFPs)

by

Charles P. Satterthwaite
United States Air Force
Wright Laboratory
Avionics Logistics Branch, WL/AAAF Bldg 635
Wright-Patterson AFB
OH 45433-6543
United States

1. SUMMARY

The ability to accurately test a system which you are developing is a highly desirable feature in the engineering design process. The ability to model your system's environment and to exercise your system, in that environment, is also highly desirable.

Operational Flight Programs are the software programs of avionics embedded computer systems. Not only is it desirable to be able to test and model Operational Flight Programs, it is essential. The consequences of not performing accurate Operational Flight Program testing can be devastating. Some of these include premature weapon releases, erroneous flight instrument displays, and complete system failure.

In order to test Operational Flight Programs, there are several things one must know about the Operational Flight Program, its weapon system host, its support environment, and how to generate and perform its test. This paper will address these issues as it develops a strategy to test an Operational Flight Program.

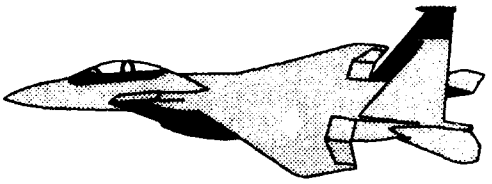


Figure 1

2. INTRODUCTION

Embedded computers are increasingly called upon to provide high-tech solutions to complex multiple threat environments for today's generation of weapon systems (Ref 6). The empowering of an embedded computer is its software, which is the Operational Flight Program.

In understanding the role of an OFP, one must thoroughly understand the threat, the weapon system, the mission, the embedded computer system, and the complex testing issues associated with OFPs (Ref 7).

The ultimate success of an updated Operational Flight Program is that the new OFP becomes an operational version. Although several layers of testing must be successfully passed before OFPs are operationally acceptable. Flight tests are expensive, as are full-up simulations. But some confidence can be gained through evaluating the OFP through a simulation environment. The simulation environment takes advantage of real-time avionics hardware, realistic simulation software, and the adaptability of advanced technologies to provide a capability for testing the weapon system, the weapon system's subsystems and units, and the weapon system's software (the OFPs) (Refs 7,9).

Testing Operational Flight Programs requires an understanding of: how OFP architecture and processes work; how an OFP is changed; the major components of an OFP and its support environment; the OFP's interaction with its users/maintainers; OFP testing/validation issues; breadth and depth of OFP tests; and how OFP test results are analyzed and interpreted (Refs 4,7).

3. OFP ARCHITECTURES AND FUNCTIONS

The Operational Flight Program literally is the software portion of a embedded computer system. The computer and its peripheral interfaces make up the system hardware. The hardware enabled by the OFP software describes the whole system.

The OFP is made up of a series of modules which represent the functions of the weapon system. These functions describe the mission phases which the weapon system can perform. Mission phases include preflight, takeoff/time to cruise, outbound cruise, SAM (surface to air missile) evasion, descent, penetration, bomb delivery, climb, air-to-air combat, inbound cruise, loiter, and approach and landing. Function types include communication

(external/internal), IFF (identification friend or foe), navigation, guidance, steering, control, target acquisition/identification, stores management, weapon delivery, and threat warning. The modules of the OFP include executive, control and display, air-to-air, air-to-ground, navigation, communication, heads up display, vertical situation display, gun, missiles, overload warning, and visual identification. A module type, such as controls and displays, might contain multiple modules which are prioritized according to the timing requirements of the functional calls of the OFP. The OFP is required to process real time interrupt driven schedules, which are handled by the executive modules. The modules of the OFP are made up of machine level object code. Access to this object code by OFP maintainers is through a higher order language source code which can be compiled to the object code. Examples of higher order languages used in maintaining OFPs are Ada, COBOL, and FORTRAN (Refs 2,6,7,8).

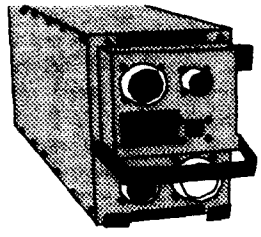


Figure 2

The embedded computer system (see Figure 2) has partitioned memory which is filled with some type of machine level object (binary) code. The OFP is loaded into this partitioned memory, and when enabled, empowers the whole system to perform its desired functions. Each embedded computer system has an instruction set which is burned into its Read Only Memory (ROM). The instruction set allows the embedded computer maintainer access to the OFP as well as the capability to optimize the remaining partitioned memory. The level of sophistication of a embedded computer system is a function of the programming expertise of its OFP maintainers, its instruction set, its memory, its hardware, and its throughput (Ref 7).

4. HOW IS AN OFP CHANGED?

Given a working OFP in a working system, why would changes ever be necessary? One reason is that the users of the system require an altered mission. As an example, a pilot would request a clearer display under some dynamic threat condition. Another reason to change OFPs is that some flaw is discovered while the embedded computer system is operational. Some combination of events might cause partial or total system failure, prompting a review and redesign in the effected areas of hardware, software, or both.

Given the task of changing an OFP (making a new version or even a new block cycle), several steps are followed to bring about the change. First, the requested change(s) is diagnosed so that it is clearly understood. Once the OFP maintainer thoroughly understands the change request, an analysis is made of the OFP areas which need to be altered. Usually the OFP is made up of a series of modules with specialized functions. A typical change might impact three modules of an OFP which contains 40 modules. The OFP maintainer will next isolate these modules by making copies of them and implementing design changes to the copies. The OFP maintainer integrates these modules by linking them together with the other unaltered modules to form a unique OFP. The OFP maintainer's final task is to thoroughly test this modified OFP by putting it through an acceptance test procedure. For a sizable OFP with several changes, a number of OFP maintainers would follow these procedures simultaneously, and then a lead OFP maintainer would integrate and test the new OFP (Ref 7).

5. MAJOR COMPONENTS OF OFP TESTING AND DEVELOPMENT

5.1 The Target Processor

In order to perform various levels of testing on OFPs, the OFPs embedded computer (also called the target processor) must be available and accessible. The actual target processor (see Figure 2) is often used by OFP maintainers to build a mockup support environment by which they can access and test their OFP changes. When these target processors are used, an environment has to be available which stimulates the processor input requirements and receives the processor output. Some examples of inputs are power, cooling, and peripheral interfaces (such as pilot commands and avionics suite inputs). Examples of outputs include pilot displays, as well as, command and control logic for other processors (Ref 7).

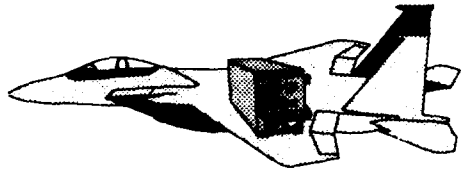


Figure 3

5.2 The Support Environment

In order to maintain an OFP, the maintainers require a dedicated computer system and a simulation environment.

The dedicated computer system (see Figures 4 and 5) allows the maintainer to access the OFP's object code as well as to copy and alter this code. The simulation environment allows maintainers to run the OFPs which enables them to interactively debug and test.

The dedicated computer system provides system conventions which are configuration management, security procedures, and proper operation of the dedicated computer system.

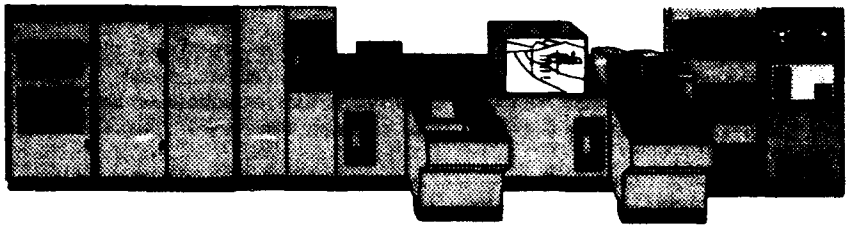


Figure 4

The hardware of a dedicated computer system usually includes mainframe computers (or powerful engineering workstations), various types of printers, disk storage devices, networking, and several access terminals.

Embedded computers and dedicated computers are frequently confused as being the same. These are actually quite different. The embedded computer is the target processor which is part of the weapon system. The dedicated computer is outside of the weapon system and is used to support the software run on the embedded computer system.

5.3 Simulation Environment

OFPs must have a means by which to operate in real-time, that is, loading them up in their target processor and exposing them to the range of conditions (or a reasonable subset of those conditions) encountered while operational. This allows the maintainer to actively debug the OFP. The degree of complexity of the OFP's environment is directly related to the complexity of this simulation environment. In the case of a typical fire control computer, a method to represent the full-up avionics suite and the dynamic environment of the fighter is required. An interface to all cockpit controls and switches, as well as, an interface between the dedicated computer system and the simulation environment is necessary. Finally, competent maintainers,

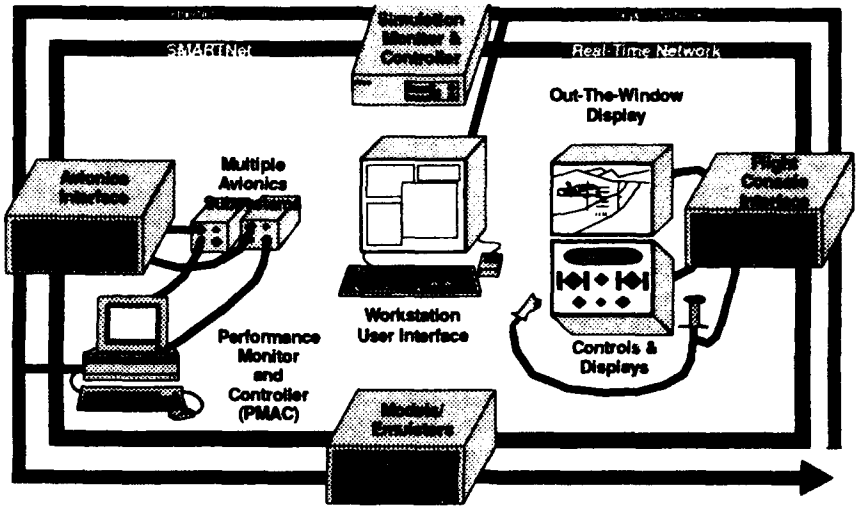


Figure 5

who know how to make the system work, are essential.

The simulation can range from a fully operational weapon system (flight testing is very expensive) to an all-software engineering workstation. Usually the simulation is a representative set of the weapon system's LRUs (Line Replaceable Units) with software emulating the cockpit and the dynamic environment.

Interaction with the simulation environment is through the dedicated computer system. Simulation utilities hosted on the dedicated computer system allow the loading of an OFP into its target processor and also allow the OFP to be exercised dynamically or statically. These utilities also allow recording, patching, debugging, freezing , and the initialization of the OFP (Refs 2,6,7,8).

5.4 The Avionics Integrated Support Facility (AISF)

The facility which houses the dedicated computer system(s) and the simulation environment(s) is the Avionics Integrated Support Facility (AISF). Another name for the AISF is the Centralized Software Support Activity (CSSA). The AISF supports one or more embedded computer systems and the associated OFPs.

6. OFP TESTING ISSUES

6.1 The Requirement to Test

The requirement to test is related to the confidence desired of the targeted system or subsystem. Low level testing might be sufficient for minor operational adjustments such as flight-line data entry. But processes affecting life support, terrain following radar, and navigation, to name a few, require highly integrated testing. These processes often require specialized testing which depend on critical resources such as specialized hardware, test equipment, test software patches, and OFP maintainer expertise (Refs 1,2,4,5,6,7,8,9).

6.2 What Is An Acceptable Level Of Testing?

This question is best asked of the crew members of the OFP's weapon system since it is their task to complete missions, as well as, survive. The quality and quantity of OFP testing affects their lives. At a minimum, crew members must be assured of the normal operating conditions of their weapon system. Additionally, maximum performance capabilities should be made available, as well as, a fail safe capability (Refs 2,8).

6.3 Iterative Nature Of OFP Testing

Usually OFPs are not acceptable in their first cut, even when they go through Operational Test and Evaluation. Five or six cycles through the testing process is not unusual. Much of this is related to the complex nature of OFPs, poor interpretation of OFP engineering change requests, and changing mission requirements midstream in OFP development (Ref 6).

7. TYPES OF OFP TESTS

7.1 The Acceptance Test Procedure

The OFP maintainers primary test is the acceptance test procedure (ATP). This test is designed to check out an OFP to a degree that it can be released with confidence to flight test and then operational test and evaluation.

The ATP is a chronological check of the OFP's responses to inputs. Inputs include switch positioning, preset conditions such as altitude or airspeed, and hardware interrupts to name a few. The OFP is loaded into its embedded computer, hosted on its simulation environment, and required to respond to these inputs in the form of static or dynamic displays, which can be checked against expected results.

The ATP for a typical fire control computer could contain 200 or more independent tests of varying degrees of complexity. The reliance of an OFP acceptance test procedure to be visually verified and to be manually performed requires several weeks to complete (Ref 7).

7.2 The Baseline Acceptance Test Procedure

The baseline acceptance test procedure (ATP) is the ATP which complemented the most recent version of the OFP (the last block cycle change). An ATP should be developed concurrently with its OFP. That is to say, any additions, deletions, or modifications to the OFP should be paralleled by the ATP (Ref 7).

7.3 Unit Tests

A unit test is the lowest level of testing. With respect to an OFP, a unit test is at the module level. As an example, there might exist some type of looping mechanism within a module. The check of this loop might be with a clock to time the loop or a count down mechanism to track the number of loop iterations (Refs 7,9).

7.4 Subsystem Tests

A subsystem test combines units to represent a functional set of an OFP. In typical fire-control computers, these subsystems might include the set of air-to-air modules or the set of control and display modules. Checks for these types of subsystems include setting a value in one module, running the OFP, and inspecting values in other modules against expected values (Refs 2,7,9).

7.5 Integrated Tests

Integrated testing, as seen in Figure 6, can represent several layers of OFP testing. Integrated testing includes the exercising of an OFP's complete module set. It is here that the subsystems are checked out against each other and against the OFP's target processor environment. The integrated test can become increasingly complex as the environment is more dynamically modeled. An example of an increasingly dynamic environment is changing from

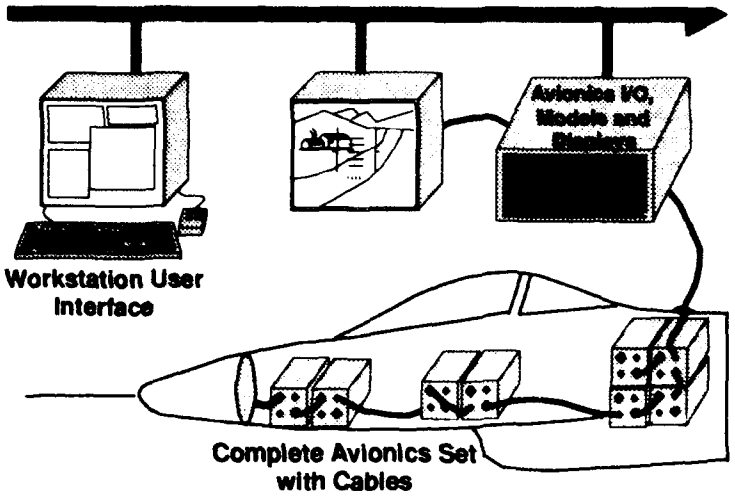


Figure 6

modeled radar inputs to actual radar inputs being driven by a separate radar OFP (Refs 2,3,6,7,8).

7.6 Static Tests

Static tests are tests which are not time dependent. Given an input, or a combination of inputs, there should be an expected response. As an example, in gun mode, a gun reticle should appear on the pilot displays. The gun reticle is a circle displayed to a pilot on a Heads Up Display (HUD) and a Visual Situation Display (VSD). The static test is that when the gun mode is initiated, the gun reticle is or is not present. If it is not present, it has failed the test.

7.7 Dynamic Tests

Dynamic tests are much more complicated than static tests, since they are time dependent. They might require a sequence of inputs over some time interval in order to ensure proper functioning of the OFP. An example of a dynamic test is to observe an expected Signal-to-Noise Ratio (SNR) improvement, as range decreases on a target being tracked with radar. The difficulty of this test is that it requires an OFP maintainer who can visually verify the test case. The maintainer has to know from experience what a sequence of responses should indicate. The quality of OFP testing in the dynamic cases is often limited to the experience level of OFP maintainers available for testing (Refs 2,7).

7.8 Classified Tests

Arrangements must be made for classified testing of OFPs. This requires the facilities and maintainers to be cleared to the level of the classification of testing. It also requires a means of properly storing and maintaining classified testing documentation. It is often convenient to isolate classified portions of OFP testing, so that non-classified OFP testing can be accomplished with minimal restrictions (Ref 7).

7.9 Automated Tests

As the complexity of OFPs increases, the ability to manually perform acceptance test procedures (ATPs) decreases. Also, the ability to fully and accurately test OFPs decreases. One successful method to increase the OFP maintainer's ability to test OFPs is to utilize automated techniques. For example, if in the process of manually

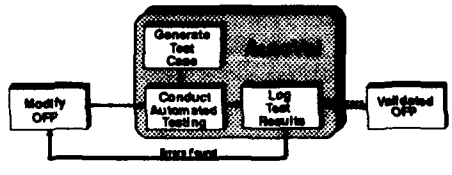


Figure 7

running an ATP test case, a sequence of switch and dial position can be captured throughspecial test software, then that portion of the ATP test case can be automated. Using techniques like this should reduce errors and the time to run through ATP and free OFP maintainers to develop more comprehensive test cases (Refs 2,3,4,5,6,7,8,9).

7.10 Operational Test And Evaluation

Operational test and evaluation is where the OFP must meet the approval of those who will use it. These users have their own check-out procedures which can include live firing of munitions, lock-on and destruction of drones, navigational exercises, to mention a few. Operational test and evaluation is the final test of a complete weapon system being fully integrated together. Several different OFPs can be evaluated during operational test and evaluation. Operational test and evaluation often finds system and subsystem errors, which were undetectable in the OFP maintainer's simulation environment. Often, OFP version updates are refined by OFP maintainers through the information they receive from Operational test and evaluation (Refs 7,9).

7.11 Developmental Test And Evaluation

Sometimes during the block cycle or version update of OFPs, a more dynamic environment than the Avionics Integrated Support environment is required. Some test situations can only be examined through the actual exercising of the complete system in its real environment. Developmental test and evaluation provides OFP maintainers with this option, usually through the provision of instrumented flight test aircraft. These instrumented aircraft can accommodate specific tests in an actual operational environment. An example of this is the recording of narrow band and wide band data in an air-to-air engagement scenario which can be analyzed for specific OFP performance parameters (Refs 7,9).

8. HOW IS AN OFP TESTED?

8.1 What is Normal?

Before originating or extending the OFP's acceptance test procedure, a baseline must be established which outlines the system's normal performance parameters and the environmental conditions which the system will experience. This baseline will influence the testing design decisions throughout the weapon system's lifecycle. In this baseline, design considerations must include the weapon system's embedded computer systems, their OFPs, and their interaction.

Performance parameters include all of the avionics of the system such as altitude, air speed, angle of attack, directional indication, and engine thrust. Performance is also the ability of the air crew to interact with the system through controls and displays. Performance also includes the system's interaction with its environmental conditions through the use of its communications, navigation, radar, electronic warfare suite, and weapons. Consideration should be given to the performance of the system's OFPs. Are the OFPs operating optimally? Are there unused resources that can be better shared? Are there potential bottlenecks or failures that can be avoided?

Environmental conditions are those situations that the weapon system will be exposed to. In the normal course of a mission, what does the weapon system experience? The weapon system is prepared for its mission at its home base. It leaves its home base enroute to its mission, it is refueled enroute, it maneuvers to avoid threats enroute, it performs its mission, and it reverses its enroute to return to home base. Several environmental conditions have been identified in this mission scenario. First, a maintenance or mission preparation environment is identified. Second, a navigational environment is pointed out. Third, a friendly air-to-air refueling environment is called for. Fourth, a threat environment is shown. Fifth, the mission performance environment occurs. And finally, there is the return environment.

In each of the above environments (plus several others), every possibility of weapon system configuration must be identified. The OFP's influence on every weapon system configuration, and subsystem configuration, in every environment in response to the weapon system's performance parameters gives the foundational basis for the OFP acceptance test procedure. The baseline OFP acceptance test takes into account every parameter, every environmental situation, and any combination of parameters and environmental situations to generate test cases which exercise these various situations (Refs 1,2,3,6,7,8,9).

8.2 What could Impact Normality?

Given a comprehensive understanding of the system's performance and the various environments in which it can be exercised, what changes, threats, or failures should be anticipated?

One of the greatest benefits of using embedded computers and software in weapon systems is that these systems can be reconfigured and adapted to changing mission requirements and evolving threats more readily than older hardware intensive systems. There is a cost associated with this benefit. In a highly integrated weapon system, small changes can effect large testing areas. It is important to know, before changes are made, how these changes influence the entire system, and what changes in testing need to be made to facilitate them.

The threat environment is constantly changing. It is necessary for weapon systems to be carefully tuned to certain threats in order to defeat or avoid them. What happens when a unique unanticipated threat is put up against the weapon system? If possible, unique threats should be anticipated and planned for in testing scenarios. Evolving and break-through technologies often translate into new threats. By keeping pace with these new technologies, potential threats can be included in the test plans.

System and subsystem failure should also be considered when anticipating potential impacts on normal testing. At what degraded capability could the system operate if various subsystems were disabled (Refs 1,2,3,6,7,8,9).

8.3 Generation of the An Acceptance Test Procedure?
Having established normal and abnormal performance criteria of the system, a comprehensive acceptance test procedure can be established. This test would begin by identifying and describing every possible configuration of the weapon system against every possible environment that the system would encounter. This test would then identify, describe, and anticipate every abnormal situation which could impact the system and its subsystems. With the inventory of configurations derived, a set of test cases would then be generated to exercise these configurations. The actual utilization of these test cases would determine the requirements for each test case such as the static or dynamic testing, degree of integration with other OFP components, simulation resources, and the number of iterations of the test case. The compilation of all this information is the acceptance test procedure.It should be noted that using present techniques to complete an acceptance test procedure, as described for a modern weapon system, would take several man months, with many of the configurations untestable (Refs 1,2,3,6,7,8,9).

8.4 Passing The Test?
What qualifies an OFP as passing its acceptance test procedure? The obvious answer is, you pass when you sufficiently satisfy every test case in your acceptance test procedure.

Because the maintenance of OFPs has not been prioritized in the procurement process, what is used for an acceptance test procedure is greatly stripped down from what has been suggested. Current OFP acceptance test procedures are heavily dependent on the OFP maintainer's subjective experience. The passing or failing of an OFP acceptance test procedure is based on how these OFP maintainers feel about their weapon system. Though not scientific or repeatable, this has been sufficient to field reliable systems.

Future OFP acceptance test procedures will demand identifiable and repeatable processes in order to guarantee weapon system reliability. The increase in configurational situations alone will disqualify the subjective expert method of passing OFP acceptance test procedures. Future OFP acceptance test procedures will require a comprehensive identification, description, and anticipation of the situations the system will and might experience. In addition, future OFP acceptance test procedures will need methods to test these situations.

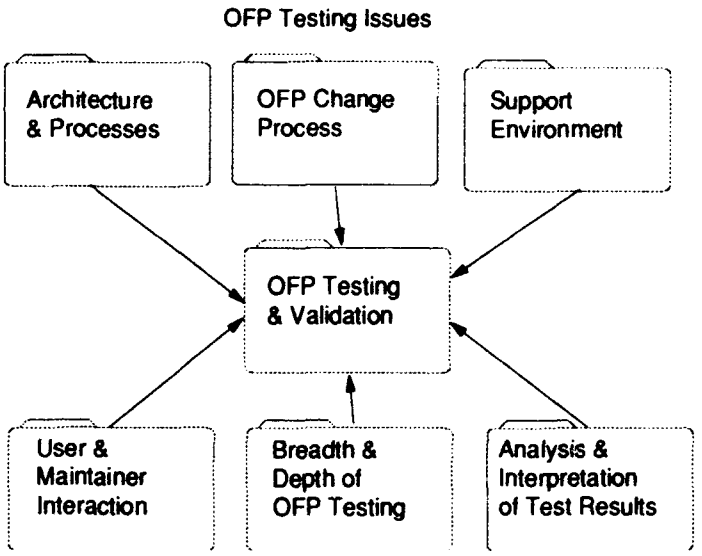


Figure 7

8.5 Increasing Levels of Integration
The nature of weapons platforms is to increase in complexity. The ability to increase in complexity has been largely facilitated by using embedded computer systems and software. These embedded systems are increasingly linked together (or integrated) to take advantage of shared resources. The consequences of increased integration is increased complexity in the ability to test the weapon

system. When subsystems are isolated, changes in those subsystems have little or no impact on the overall weapon system. When these subsystems are integrated through some shared resources, changes in a subsystem potentially impacts all of its sharing partners.

Unfortunately, as systems have become more complex, the capability to test these systems has not kept pace. This is largely due to the fact that the procurement process has not provided for or anticipated the maintenance requirements of advanced avionics software. It is well documented that 70% or more of a system's life cycle cost will be in the maintenance of that systems software. A large portion of this cost lies in the system's testing (Refs 7,9).

For every increased level of system integration, at least equal thought, design, and resources should be dedicated to testing. This will require new analysis, methodologies, and testing techniques (Refs 2,3,4,5,6,7,8).

8.6 Need for Advanced Technologies

In order to assure the successful operation of current and future avionics weapon systems, as well as, the growing number of system platforms implementing highly integrated embedded computer systems and software, advanced avionics testing technologies must be encouraged and accelerated. Some of the areas to be pursued include: improved instrumentation techniques; development of integrated diagnostics techniques (especially in the area ofsoftware integrated diagnostics); continued emphasis on automated testing techniques; development of advanced verification and validation techniques; expansion of avionics software reuse libraries; improved simulation and testing environments; increased implementation of hypermedia and virtual reality technologies into the OFP testing environments; and continued development of human factor engineering (Refs 2,3,4,6,7,8).

The encouragement and implementation of these types of technologies will: enable the weapon system to monitor itself while it is operational; return from its mission and give its maintenance staff a comprehensive performance and diagnostics report; suggest new techniques for evaluating complicated highly integrated OFPs; and identify reserve capabilities and opportunities for the weapon system (Refs 2,8).

9. CONCLUSIONS

Operational Flight Programs hosted on embedded computer systems have greatly extended the capabilities of avionics weapon systems. These extensions have increased the weapon systems lethality; the air crews survivability, and the capability of the system to be reconfigured as well as decreased the weapon system turn around time In order to be further extended, a new emphasis must be placed on the testing of Operational Flight Programs. This new emphasis is dependent on the inclusion of advanced avionics technologies into existing and planned Avionics Integrated Support Facilities. It is also dependent on all individuals involved in the acquisition and maintenance of weapon systems containing OFPs to be aware of what it takes to have confidence in the software.

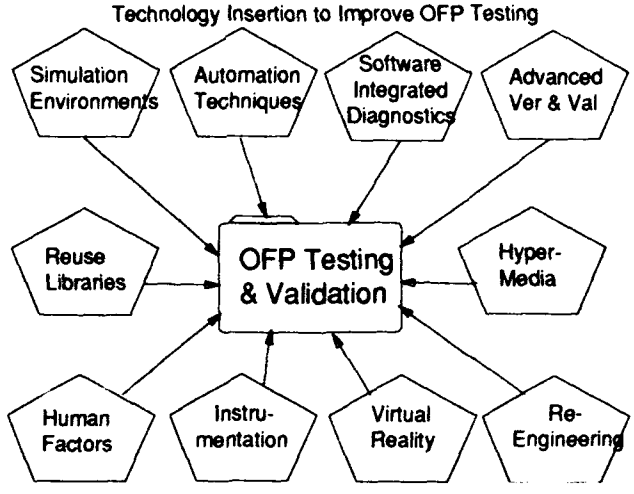


Figure 8

10. REFERENCES

[1] DOD-STD-2167A, Military Standard Defense System Software Development, 29 February 1988.

[2] Blais,R.R.,Neese,R.E.,Noharty,,K.L., "Modular Embedded Computer Software (MECS) Final Report, Volume II", Air Force Technical Report WL-TR-92-1101, Wright-Patterson AFB, OH, June 1992.

[3] Harris,R.L., "Laboratory Concepts in Avionics Software", IEEE/AIAA/NASA 9th Digital Avionics Systems Conference Proceedings, 15-18 October 1990, Virginia Beach VA.

[4] Harris,R.L.,Jackson,O., "Software Engineering Tools For Avionics Embedded Computer Resources", IEEE/AIAA National Aerospace Electronics Conference Proceedings, 21-25 May 1990, Dayton OH.

[5] Jones,J.V., *Engineering Design Reliability, Maintainability, Testability*, TAB Books Inc., Blue Ridge Summit, PA, 1988.

[6] Morris,D.M., "Avionics Operational Flight Program Software Supportability", IEEE/AIAA/NASA 9th Digital Avionics Systems Conference Proceedings, 15-18 October 1990, Virginia Beach VA.

[7] Satterthwaite,C.P., *Maintaining An Operational Flight Program (OFP)*, Air Force Technical Memorandum WL- TM-91-123, Wright-Patterson AFB, OH, 1991.

[8] Satterthwaite,C.P.,"Getting a Handle On Designing For Avionics Software Supportability And Maintainability" IEEE/AIAAA National Aerospace Electronics Conference Proceedings,18-22 May 1992,Dayton OH

[9] Computer Resource Acquisition Course, "Course Materials Volume I and II", System Acquisition School, 6575TH School Squadron, Air Force Systems Command, Brooks AFB, TX, December 1991.

Discussion

Question C. ANTOINE

How will the use of Ada and OOD impact the OFP testing, in particular the flight testing activities?

Reply

The immediate and near future of Ada and OOD will have little impact on OFP testing because of the enormous amount of non standard code in use in the inventory.

What these can impact now is improve software engineering practices amongst those developping and maintaining software. When disciplined software becomes standard in development, huge opportunities will exist, because not only re-usable software will be available accross weapon systems. This will include support environment, software and test cases.

OOD most certainly will enable re-use libraries, which will also avail new opportunities. Note that flight testing is hugely expensive and weapon system dependent. These alone hinder absorbtion of new techniques.

By the way, F-22 will have many challenges in OFP testing even with its Ada advantages.

Question K. BRAMMER

You mentionned re-use as one of the technologies with potential benefit. Re-use can be practiced at all levels of the system hierachy, from CSU level, CSC level and so on up to subsystem level. I am interested in your opinion, at which of these levels re-use would be most desirable, i.e. rather at the upper levels or at the lower ones and, may be, depending on the application domain?

Reply

I am most interested in RE-USABLE COMPONENTS at the lowest levels. I would like to see hardened proven "units" which can be "grabbed" and "used" accross dissimilar systems. Much work has to be done to design this type of software unit to be robust. Namely, techniques to quantify and qualify components.

For larger units (subsystems), todays limitations are prohibitive. It may be possible to have a common subsystem (such as a common radar) accross systems.

To make software components accountable to quality standards, we have to develop new techniques to measure and diagnose software dynamically. This capability does not exist. I propose rigorous new research in software integrated diagnostics which will begin to enable QA concerns with information to assure software products. One component necessary for software integrated diagnostics is the R3000's Built-In-Support function, which is an intrusive (has a cost) hardware/software way of observing and exercising your system.

Question J. BART

Are things getting better, eg with F-16?

Reply

The F-16 SPO traditionally was held to a single option. They jumped at the opportunity to have a choice. Our Advanced Multipurpose Support Environment (AMPSE) has been instrumental in allowing the F-16 A:B models to be updated timely. The new generation of sim is also being adopted (COMET), as well as being a BGTA test site for many technologies including Autoval (automated validation), PMAC (programmable monitor and control), to name a view.

Unfortunately, the OFP work load of systems like F-16 and F-15 AISF's are inceasing beyond what current techniques can handle. Many opportunities exist for technology transition. The greatest need is to leverage technology before crisis situations dictate continued band aide usage.

We need to pry loose maintenance people to steer and transition technology.

Integrated Formal Verification and Validation of Safety Critical Software

N J Ward
TA Consultancy Services Limited
'The Barbican'
East Street
Farnham
Surrey GU9 7TB

SUMMARY

Embedded software providing the functionality for a flight vehicle self destruct system was judged to be safety critical and required the highest level of assurance in its correctness. In order to achieve this a programme of independent verification and validation was initiated which involved the definition of a formal specification of the software combined with static analysis and dynamic testing. The formal specification, written in an Object Oriented form of Z, was used to clarify the requirements and to provide a definition against which the code could be formally verified. A range of static analyses were performed, culminating in Compliance analysis which effectively provided a proof of the code against low level mathematical specifications that were refined down from the Z specification. The dynamic test sets were chosen partly from the requirements specification and partly from the static analysis results so that complete path coverage through every module was achieved. The work revealed a number of errors within the code and its specifications, which were corrected. Through its rigour and the identification of errors the analysis has given a very high degree of confidence in the correctness of the software.

1. INTRODUCTION

Safety Critical software is regarded as that for which a software failure could lead to loss of human life. The integrity of such software is therefore a subject of great concern for developers, users and the general public. To reflect these concerns there is currently a proliferation of standards across a wide range of industries. At a European level two draft standards have recently been produced under IEC 65A working Groups 9[1] and 10[2] and there are a number of sector-specific industry variants of these in production. Other industry sectors have developed their own standards separately, for example IEC 880[3] in the Nuclear Sector, DO-178A[4]/B in Civil Aviation sector and Interim Defence Standards 00-56[5] and 00-55[6] in the UK defence arena.

All these standards have the same aim, namely to ensure the correctness and safety of software developed under them. In general the approach defined within many of the standards is the same, comprising the initial conduct of system hazard analyses leading to the categorisation of the software into criticality levels. Techniques, methods and requirements are then defined for both the design and verification of the software within each criticality level.

One notable area of difference is in the avionics sector, in particular between Civil avionics and UK military avionics. In the former area the emphasis, in practice if not in mandated requirements, has been on the use of techniques such as software diversity as a means of trying to eliminate the effects of single software faults. However, even using different teams in different companies and different languages/hardware platforms, there is no guarantee that the same mistake could not exist in more than one implementation. Furthermore, there is often a common requirements specification which is a potential source of common mode failure.

Perhaps in recognition of these problems, and perhaps in recognition of the extra cost that diversity brings, the emphasis on the UK military side has been on 'doing it once and doing it right'. Particular emphasis has been placed on mathematics and analysis, and a very proscriptive standard, Interim Def Stan 00-55 has been produced which mandates the use of Formal Method techniques.

Formal Methods techniques have been shown on particular projects and in particular application areas to have significant benefits in terms of safety and correctness of software. There is, however, some debate about their universal appropriateness, in particular for non trivial systems. For these and other reasons formal methods are understood to be covered in less than one page in DO-178B, which is a significant contrast to Def Stan 00-55 which is largely based around the concepts of formality and proof.

Whilst not able to contribute to the debate on suitability of formal methods for large systems, this paper describes a project where such methods, combined with static analysis and dynamic testing, were used successfully on a small embedded software system.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

2. DESCRIPTION OF APPLICATION

The particular application was a flight vehicle self-destruct system. This system was to be fitted to the flight vehicles during range testing and was to permit the destruction of the vehicle if any problems arose during trial flights. The system was therefore judged to be the highest level of safety criticality, since any failure to destruct when required could permit the vehicle to fly outside the range and potentially cause harm or damage to people and property.

The self destruct system itself was relatively simple in operation. Its overall function was to receive a radio signal from the ground control system and, if appropriate, to set off a small explosive charge which would lead to in-flight destruction of the flight vehicle into relatively harmless pieces. The main functionality of the system was provided by software, written in an 8-bit assembler (Motorola 6805) and comprising approximately 500 lines in total.

The software was required to perform a number of individual functions. Firstly it had to receive and decode the communication signal. Secondly it had to interpret the signal and decide whether it was of relevance for the particular flight vehicle (the system allows for a number of identical vehicles being used on a number of different ranges at the same time). Finally it had to decide, based on the signal received and on timing information, whether it was necessary to initiate destruction.

The fail safe action of the system was to cause destruction and hence there were a number of design features which aimed to ensure, for almost all possible failures, that destruction will take place. For example, the ground system itself should always be issuing either a fire (destruct) or prohibit signal. As well as destruction being initiated by the appropriate fire signal, a timer within the software should also initiate destruction if a prohibit signal has not been received for a pre-defined length of time (for example if communication from the ground is lost). Finally there was also a hardware element to pick up the case of a software 'lock up' where a monostable would cause initiation if it had not received a periodic reset signal from the software. Despite all these precautions the software was still considered safety critical since a catastrophic logic error could be possible which, for example kept the timers initiated but failed to act on a fire command.

The software was developed 'conventionally' from a top level, natural language, system requirements specification down through a software requirements specification and software design specification, below which was the source code itself. The low level detailed specifications for each section of code were provided as code headers. No particular structured methods or tools were used during the software development process.

3. OBJECTIVES

It is often the case that awareness of the issues relating to software criticality occurs relatively late in a project. For example, it may not be appreciated at the start of the project that the software is safety critical and therefore activities will not be planned which will help the certification of the software. This is likely to result in additional costs at the end of the project which could have been saved if the issues had been addressed at the start.

To some extent this was the case with the software for the flight vehicle self-destruct system in that TA Consultancy Services were first approached following the completion of the initial version of the software. TA Consultancy Services were requested to ensure that the software was acceptable in a safety critical application. Analyses had already been carried out by the system/software manufacturer which confirmed that the software was safety critical. It was therefore necessary for the software to be approved or certificated as being safe to use before the first flight of the system.

TA Consultancy Services were requested to conduct analyses and Independent Verification and Validation (IV&V) to increase confidence in the correctness of the software. Although no standards had been specifically mandated for the software to meet, it was also the aim for TA Consultancy Services to carry out any retrospective work necessary to enable the software to meet the spirit, if not the letter, of Def Stan 00-55, which was at draft stage at the time.

4. OVERVIEW OF APPROACH

At a very simplistic level there are two essential stages to ensuring the correctness of a piece of software. The first is to ensure that the requirements specification is correct and the second is to ensure that the code completely meets the requirements specification. Unfortunately neither of these activities is easy. For example the task to ensure that the requirements are correct is a largely unbounded problem necessitating effective communication with the potential users. For a non-trivial system it is very difficult to envisage all the specific requirements without extensive modelling and simulation of the system and the unambiguous and complete representation of the requirements is another problem in itself.

Similarly the task of ensuring that the code meets its requirements becomes much more problematic with increasing size of the software. The development of software for any non-trivial system will involve a number of stages of specification and design refinement. It is therefore not practicable to verify or validate the code directly against the requirement in a comprehensive and complete manner. Consequently this objective has to be achieved by verifying each stage of design refinement against the previous level.

The approach taken on this project was effectively that defined under Def Stan 00-55. In particular, to meet the first requirement of ensuring the correctness of the requirements

specification, it had been decided to use formal methods of software specification. To meet the second requirement of ensuring that subsequent stages of development meet the preceding ones it was decided to use static analysis to verify, as rigorously as possible, that the code ultimately met its requirements. Dynamic testing was also to be used, at a unit and system level, to provide the final demonstration of functionality of the final embedded code. Overall, however, what was planned was to integrate all three of these activities to maximise confidence in the software and to minimise the amount of effort involved in each. The following sections describe the work carried under each activity.

5. FORMAL SPECIFICATION

Formal Methods of software specification involve the use of mathematics to represent the specification for the software, typically at the requirements level. The 'raison d'etre' of such methods is that it is considered that natural language (eg English) is too imprecise and prone to ambiguity to be able to define, accurately, the required behaviour of the system. There are numerous examples of errors introduced into systems as a result of misunderstandings between users and implementors, originating from ambiguities, errors or omissions in the requirements specification.

The use of mathematics can help to eliminate such causes of errors since precise relationships and requirements are defined as a consequence. It is also argued that the use of such techniques puts Software Engineering on an equivalent level to other engineering disciplines, instead of being more of a 'craft'. Consequently a number of Formal Methods techniques have been originated and used over the last few years. There are a number of different types of such methods, for example some use set theory and predicate logic, others use algebraic equations and others are process based. They all have an agreed notation in mathematical terms, have a formal calculus for reasoning about statements and enable arguments to be constructed. The more established methods, such as VDM, Z and OBJ also have tool support for assisting with syntax and consistency checking.

On this project, the manufacturer had realised that there was a need for a formal specification if the software was to conform to Def Stan 00-55. They had therefore produced their own specification written in Z. This had been produced after the implementation by someone who had been on a Z course but was not a formal methods expert. The outcome was a specification that was not required for progressing the design (since this was already completed) but was also not particularly suitable for verification and proof. The first task for TA Consultancy Services was therefore to produce a new Z specification.

It was decided to produce the Formal Specification in a object oriented form of Z originated at Surrey University. It was decided to use this particular method, known as Object Orientated Process Specification (OOPS)[7], firstly because of TA Consultancy Services' familiarity with it but also because it allows Z specifications to be smaller and more manageable than might otherwise be the case. One of the features of OOPS is that it possesses a 'rest unchanged' rule whereby, unlike standard Z, one does not have to re-state all those relationships that have not changed since the previous expression. It was considered that this would facilitate subsequent proof of the code against the specification.

There were no tools available to support OOPS so none was used during the development of the formal specification. However, since at the time of conducting the work there were few, if any, Z-based tools available (eg syntax checkers) for any other variants of Z, this was not considered to be a significant factor in the choice of Z variant.

The OOPS specification was based on the original Z specification and was structured to follow the original requirements, rather than try and be structured in a way that would reflect the design and implementation of the code. Although it was appreciated that this would not assist in the 'proof' of the code against the specification, it was considered that structuring the Z specification to follow the requirements would facilitate the initial stage of ensuring that the formal specification accurately reflected the stated requirements for the system. Where possible, though, objects were chosen that reflected the known architecture of the implementation if this was considered likely to facilitate the subsequent analysis without compromising the 'strict' representation of the requirements.

The next decision to be taken was the level of abstraction of the specification. Typically formal specifications at a requirements level should be produced at a high level of abstraction and then progressively refined down through the design, adding implementation detail, until the code level is reached. In this case, because it was a very small system (500 lines), and because refinement into a design was not required, it was decided to produce the OOPS specification at an intermediate level of abstraction. This would provide an ability to verify or prove the code against the specification with the minimum amount of levels of refinement necessary.

The OOPS specification document was produced containing the formal Z schemas and an English language commentary. Defence Standard 00-55 calls for animation of the formal specification in order to 'validate' it, however, it was considered that this was not necessary in this case because of the small size of the system. Instead the OOPS specification was validated by a series of formal review meetings held between the software manufacturer and TA Consultancy Services.

The production of the formal specification and the reviews of it were found to be useful in clarifying details of some aspects of the requirements and in raising some potential problems. In particular the software performed a number of time dependent activities to enable it to obtain synchronisation with the controlling signal. The review activities proved to be useful in clarifying details of the required operation of this part of the software and also suggested that this could be an area where the implementation may have difficulty in meeting the requirements.

Once it had been produced and reviewed as being 'correct' the formal specification was then used as the initial point for the Compliance Analysis described under static analysis below.

6. STATIC VERIFICATION

6.1 Background

The UK Ministry of Defence has, for many years, required the use of static analysis on safety critical airborne software. This requirement originated from concerns in the late 1970s that dynamic testing was insufficient to provide complete confidence in the correctness of software. This was because of the known inability of testing to provide complete coverage of all possible paths through any non-trivial piece of software. Testing can therefore only show the presence of faults, not their absence.

As a result of this concern, research was commenced into methods of analysing software, using mathematical techniques, initially to investigate the structure of the code. A number of tools were produced from this research, one of which was MALPAS. The aim of these static analysis tools was, using analysis rather than execution, firstly to reveal potential errors in the software and secondly to permit the rigorous verification of code against specifications.

6.2 What is Static Analysis?

Static Analysis is a very widely used term which means different things to different people. At its simplest level, static analysis is the examination of code without the execution of the code. In such terms even code walkthroughs can be considered as a form of static analysis. More typically static analysis is performed by software tools but these again provide a wide range of different types of analysis. In broad terms, however, it is possible to identify four main types of analysis namely Metrics Analysis, Flow Analysis, Semantic Analysis and Compliance Analysis. There is by no means universal agreement on these categories or these names but, for the purposes of this paper one can define each as follows:

6.2.1 Metrics Analysis

This form of static analysis is performed by a number of well known tools and aims to provide figures that give an indication of the 'quality' of the source code. Typical figures that are produced include the number of lines in each program section/procedure, the number of paths through each program section, the ratio of comment lines to executable lines and figures defining the complexity of the code constructs. Sometimes the tools are also able to identify and flag usage of particular code constructs that can help in ensuring that coding codes of practices have been followed.

Such analyses tend to be quick, and therefore cheap to perform. Whilst they give an indication of the broad quality of a piece of code, this is no guarantee at all that the code is correct. Indeed substantial judgement may be required in interpreting the results. For example there may be good reasons why a particular procedure is many times longer than the ideal length of around 50 lines of source code. Similarly whilst it may be of interest to know which are the most complex sections of code, these may not be the most critical from a functionality point of view and by focusing attention on complexity one may be diverted from the main issues of criticality and correctness.

6.2.2 Flow Analysis

This analysis concentrates on the flow of control, data and information through the code. It reveals this to the user in an appropriate descriptive form, highlighting where appropriate, potential anomalies in each of the types of flow.

For example a graph of the control flow of each procedure is often displayed along with an identification of particular features, such as the location of loops, entry and end points. The tools may be able to provide a statement of whether the code is well structured (eg according to rules defined by Dijkstra) or badly structured. Of most importance is the identification of any unsafe features such as dynamic halts and loops with multiple entries. It may also be possible for any syntactically unreachable or redundant code to be identified.

Analysis of the data usage of code can provide information to the user that can be used to check against the required data usage as defined in a low level specification. Additionally such analysis can explicitly identify data usage anomalies that could be an indication of software errors. Typical data usage information provided includes an identification of which parameters are read and which are written and this can be checked against input/output lists defined in the specification in order to verify the correctness of these aspects of the code and specifications. The sort of anomalies highlighted can include IN parameters written to (which may indicate bad programming practice) and OUT parameters never written. Sometimes the user can define to the tool what the expected data usage is for the program section and the tool will then identify whether this is in fact the case.

Finally within this category is analysis which provides details of the information flow through the code and an indication of the dependencies between data items. For example the analysis may identify which inputs affect which outputs. Such results can be used to check, from knowledge of the specification, that there are no unexpected dependencies or missing dependencies. This sort of analysis may be most suitable for security critical code where it may be of particular importance to ensure that specific outputs are only affected by defined inputs.

Flow analysis is particularly useful for ensuring correctness of the syntax of the code, for checking that the correct inputs are passed in and out and for identifying potential errors in the flow of control and the usage of data. To some extent some of these issues are checked or protected against through the use of 'good' high level languages, such as Ada, and 'strict' compilers. Even so the numbers of anomalies revealed by such analyses can be high and can save

substantial amounts of time and effort if detected at an early stage. However, such analyses cannot check that the code meaning or semantics are correct. For this one needs to use semantic or compliance analysis.

6.2.3 Semantic Analysis

Semantic Analysis, sometimes known as symbolic execution, involves the checking of the semantics of each path through a program section/procedure. Typically sophisticated tools, of which there are few, are used to detail the precise mathematical relationship between program section inputs and outputs for each path through that program section. Usually the results are expressed in terms of a path condition, or predicate, defining the relationship between inputs necessary for the particular path to be executed. Functional relationships are then defined between the inputs and outputs for that path. Sometimes the results can be presented in a tabular form, as a 'truth table'.

The effect of semantic analysis is to reveal, for the whole range of input variables for each program section, exactly what the code does in all circumstances. The user can then check this information to ensure that the software performs correctly and meets its specification. A significant benefit of this form of analysis is that it involves turning the sequential, and potentially confusing, logic of source code into parallel logic. By showing relationships in terms of inputs and outputs many errors have typically been revealed in terms of unexpected paths, incorrect relationships and even incorrect specifications.

However, even with tools providing the representation of the code semantics on a path-by path form, there is still the need for substantial human involvement in comparing the tools output with the specification for the software and deciding whether the two agree. A reduction on the human involvement, and an increase in the tool-assisted verification of the code is provided by Compliance Analysis.

6.2.4 Compliance Analysis

Compliance Analysis aims to show, in as automated manner, as possible that the code meets its specification. In order to be able to do this it is necessary to define the specification for each program section precisely; this is usually done in a mathematical form using predicate logic. In particular the specification may be embedded in the code as PRE conditions, defining conditions on the inputs, and POST conditions defining required relationships between inputs and outputs. There is also likely to be the need for assertions entered within the code to define intermediate requirements, for example as loop invariants.

After embedding the specification in a form that the tool can interpret, the analysis is conducted using a combination of tools assistance and analyst direction. The role of the analyst is typically to provide guidance to the tool in the choice of rules and relationships that are invoked whilst showing conformance between the code and specifications. Depending on the power of the particular tool and its simplification ability, the involvement of the analysts may be very large or could be quite modest.

Compliance Analysis is effectively performing a proof of the code against a low level mathematical specification. In this respect it is by far the most rigorous of the static analysis techniques. However, this is at the expense of cost because of the potentially large amount of analyst effort involved. Typically Compliance Analysis productivity is around an order of magnitude worse than the next most rigorous form of analysis (Semantic Analysis) at around 5 lines per man day. One therefore has to decide whether the criticality of the code justifies this level of expenditure or whether a less rigorous and less costly form of analysis can be used.

6.3 Requirements

Having originated the tools that perform the most rigorous forms of static analysis, namely Semantic and Compliance analysis, the UK Ministry of Defence has requested the use of such techniques on safety critical airborne software for some years, although it has not been mandated in a standard until recently. One of the earlier standards in the MoD avionics arena mentioning static analysis techniques is Interim Defence Standard 00-31[8], produced in 1987. This standard effectively follows DO-178A but, amongst other things, also requires the use of an 'approved analysis process' for the verification of module implementation.

In practice static analysis, up to and including Semantic Analysis, has been carried out on most airborne UK military safety critical systems, almost invariably as an independent, post development, verification task. Experience has shown the benefits of the use of these techniques, even if they have represented an irrecoverable cost at the end of the development programme. Consequently there are continuing and increasing requirements for such work to be conducted on all safety critical MoD systems, to the extent that Interim Def Stan 00-55 also calls for Compliance analysis.

6.4 Analysis Conducted

In order to meet the requirements of Def Stan 00-55 on this project it was decided to carry out three main forms of static analysis, namely Flow Analysis, Semantic Analysis and Compliance Analysis. Some metrics were produced as part of this analysis but it was not considered necessary to specifically analyse and assess the 'quality' aspects of the code since formal verification would go considerably beyond any issues of quality.

It was decided to use MALPAS for this work since this is the tool with which TA Consultancy Services are most familiar. It is also more suited to retrospective verification than other similar tools. In order to perform such analysis MALPAS requires the source code to be translated into MALPAS IL (Intermediate Language) which is the input language for the tool. Often this is done automatically using one of a number of automatic translators that exist for a range of languages. However, in this case, since a translator for 6805 assembler

did not exist at the time and since there was insufficient code to be translated to justify the development of a translator, hand translation was used.

6.4.1 Translation

The initial stage of any hand translation process is to define the mappings between the source language and IL, the MALPAS input language. IL has a superficial resemblance to Pascal or Ada but also contains constructs that allow it to be used for the modelling of low level languages. Mappings were defined between 6805 assembler and IL for all the instructions and addressing modes used in the code to be analysed.

TA Consultancy Services have substantial experience in the derivation of mappings and the production of translators and this was used to obtain an appropriate compromise between abstraction and precision which would facilitate analysis whist ensuring correctness of modelling. The mappings were designed so that they permitted the checking of a number of features of assembler code (such as aliasing and overflow) that are known from experience to be the potential source of errors in the code. The mappings were defined in a mapping document which went through a number of stages of formal review in order to ensure correctness and suitability.

6.4.2 Analysis Control

TA Consultancy Services' experience with this form of analysis is that it is essential that the results are reproducible and consistent. This is particularly true of large projects where there are large numbers of analysts working together but it is also relevant for small projects such as this where analyses may need to be re-run at a later date, possibly using different people.

Consequently TA Consultancy Services have defined their own 'Standards and Procedures' for this form of work. This document, analogous to a coding code of practice for software development, defines the precise way in which the tool will be run, how the configuration control aspects will be managed and the results recorded and reported. A further feature is a requirement for formal reviews of all analysis work, the aim of which is to try to eliminate any possibility of human error in those parts of the work where manual interpretation is necessary.

Additionally, a series of forms, developed by TA Consultancy Services, were used on this task for recording the analysis results. The particular benefits of such forms are, firstly, that they permit the recording of the analyst's interpretation of the analysis results and, secondly, that they provide a valuable verification record that may be used by a certificating authority for checking on the effectiveness of the work carried out.

6.4.3 Flow and Semantic Analysis

These analyses were conducted in a bottom-up manner on a procedure by procedure basis. That is to say that the lowest level procedures/subroutines that call no others were initially analysed. These were then represented as MALPAS IL procedure interface specifications which themselves are automatically substituted in by MALPAS at the procedure call. This feature of the tool prevents an exponential build up in complexity as the analysis progresses up the calling hierarchy and permits the analyst to control the relevant information passed up to higher levels. In this particular case, because of the small size of the software there was little calling hierarchy and the overall analysis was conducted in very few stages.

The analysis was conducted a procedure section at a time and the results compared against both the intermediate level natural language specification and the embedded code comments that represented the low level specification. It could be argued that Semantic Analysis may not be strictly necessary where Compliance analysis is also being conducted since the Compliance Analysis performs a more rigorous verification of the code against specifications. However, the Compliance analysis verifies that the code meets the specification provided but will not identify whether the code also performs other functions. Semantic analysis will reveal all the functionality of the code and permit identification of any functionality additional to that given in the specification. It was therefore considered necessary to perform Semantic analysis as well as Compliance Analysis in this case.

An example output from the analysis is shown below in figure 1, demonstrating the tabular form of the semantic output. This example represents a section of code containing five semantically possible paths from the self-destruct system software. The first part of the results shown gives some examples of predicates (expressions on input variables) making up the path conditions through the code. The second section gives examples of actions (relationships between inputs and outputs for each output) for output variables written to in this section of code. Finally the paths table details the conditions necessary for each path to be executed and identifies the assignments to each of the output variables. One point of note in this example is that a number of the output variables, for example *opcomp* and *ttr*, are assigned the same expression, irrespective of which path through the code is executed.

All anomalies found during this analysis were documented, categorised and reported to the software manufacture for discussion and resolution. This aspect is discussed in more detail in section 6.5 below.

Conditions

C2: cposn
= nxtbit_set_bitctr (bitctr, count_h, count_l, opcomp_h, opcomp_l, prhbtto, time_h, time_l, tsr, ttr_h, ttr_l)

C4: pos(2) > pos (cmp6_set_ax (31, nxtbit_set_ipl (ip_l, porta, prhbtto, time_h, time_l)))

C6: afire = lstcmd

C7: frmlgth = cposn

Actions

Assignments to bitctr

A11 : nxtbit_set_bitctr (bitctr, count_h, count_l, opcomp_h, opcomp_l, prhbtto, time_h, time_l, tsr, ttr_h, ttr_l)

A12 : 0

Assignments to lstcmd

A21 : nonvald
A22 : afire

Assignments to time_h

A51 : timers_set_timeh (time_h, time_l, prhbtto)
A52 : 0

Assignments to time_l

A61 : timers_set_timel (time_h, time_l, prhbtto)
A62 : 0

Paths Table

Conditions	Paths				
	1	2	3	4	5
C1	F	F	F	F	F
C2	F	T	T	T	
C3	F				T
C4		F	F	T	
C5		T	F		
C6				F	
C7					F
C8					T
bitctr	A11	A11	A11	A11	A12
lstcmd	A21	A21	A22		
porta		A31	A32	A31	A33
portb	A41	A42	A43	A42	A44
time_h	A51	A51	A52	A51	A51
time_l	A61	A61	A62	A61	A61
ip_h	A71	A71	A71	A71	A71
ip_l	A81	A81	A81	A81	A81
ttr_h	A91	A91	A91	A91	A91
ttr_l	A101	A101	A101	A101	A81
opcomp_h	A111	A111	A111	A111	A111
opcomp_l	A121	A121	A121	A121	A82

Figure 1 - Example Semantic Analysis Output

6.4.4 Compliance Analysis

The Compliance Analysis was conducted independently of the Flow and Semantic Analysis. As has been described in section 6.2.4 above, the first task was to produce the embedded specification, in terms of PRE and POST conditions, and any necessary ASSERT statements, from the OOPS specification. The initial stage was therefore to refine the OOPS specification down to the required low level mathematical specification.

This refinement was performed manually but in as rigorously a manner as possible, with all the refinement work being reviewed. The following illustration shows how an abstract statement within the OOPS specification is refined so that it is represented in items and functions that appear within the IL translation of the source code.

This particular example of the implementation interfaces with the microprocessors hardware time system to obtain access to timing information. This is modelled in the analysis by introducing the specification variable, *instr_time*. The refinement here involves turning an abstract tolerance condition into a detailed expression at the level of assembler arithmetic and bit manipulations.

Consider the refinement of the timing condition

"val(t) - tt > = tol (125)":-

Where *tol* relates to a tolerance on the specified time in microseconds. In this case one is interested in the topmost limit, chosen to be 140μs. Refinement from this into the variables used in the code, using the knowledge that 2 cycles = 1μs, gives the following:

val(t) - tt > = tol (125)
val(t) - tt > = 140
opcomp = ttr + 140 ^ tsr ^ instr_time > = 280

sum (opcomp_l, opcomp_h) =
plus (sum (ttr_l, ttr_h), sum (70,0)) AND
bit (6, tsr) = 1 AND instr_time > = 280

As was mentioned in section 5 above, the structuring used in the design did not reflect the natural structuring of the specification. In particular, the implemented procedures often contained the behaviourial functionality of a number of operations in the specification. This complicated the analysis and means that numerous ASSERT statements had to be used to relate the code to the specification.

The MALPAS Compliance Analyser works by comparing the code against the embedded specification and explicitly identifying any differences between the two. Such differences are identified as a threat; hence if the code meets its specification the threat is declared to be 'false'. In theory this operation can be performed entirely automatically, however, because the MALPAS algebraic simplifier is not all-powerful, manual assistance is usually necessary. This assistance takes the form of producing replacement rules which define the semantics of rules that the analyst wishes to

apply. These rules may be defined directly from the formal specification semantics or may simply express a standard mathematical relationship that the simplifier is unable to recognise without assistance.

6.5 Analysis Results

The outcome from each part of static analysis was either that each program section was found to agree with its specification or that anomalies were discovered. These anomalies would range from identified code errors through to comments on how particular aspects of the code or specifications could be improved. In common with other similar analysis projects that TA Consultancy Services have conducted, the anomalies raised were categorised in terms of seriousness by the analyst who raised the problem.

Three categories of anomalies were defined, as follows:

A Technical error or omission causing non-conformance with specification which may have a significant impact on safety.

B An ambiguity or suspect design feature of lesser significance than A for which corrective action is considered desirable though not essential.

C Observation of minor significance. No immediate action is necessary.

In total 42 anomalies were raised as a result of the static analysis, of which 10 were category A, 13 were category B and 19 were category C. This level of comment is considered quite good bearing in mind the minimal verification and validation work that had been conducted prior to TA Consultancy Services' involvement. Furthermore, it is only about 50% worse (in terms of anomalies per line of code) than the rate reported by TA Consultancy Services on supposedly fully verified code on a number of other projects.

It is not possible to say what proportion of anomalies was discovered during each form of static analysis because of the order in which the analysis was conducted. For example the Semantic and Compliance analyses were conducted in parallel to some extent so anomalies may have been discovered first in one before being picked up in the other. It is possible to say that a number of anomalies relating to timing were identified and clarified more during the Compliance Analysis than during the other analyses. This was primarily a consequence of the concerns about timing that were first appreciated during the derivation of the OOPS specification. As a result of this concern additional modelling of timing aspects was performed during the Compliance Analysis and led to the problems becoming further defined.

Following the reporting of the anomalies by TA Consultancy Services, each one was discussed with the customer and corrective action agreed. Although some anomalies raised were clear cut errors or deficiencies, many (as has commonly been found in other projects) involved a degree of subjective judgement. For example an anomaly may be raised if the code was considered insufficiently defensive but, depending on the manufacturers view regarding the likelihood of an accident being caused by the lack of defensiveness, the code may or may not be corrected.

One such example raised concerned a timer routine which checked to see whether the timer had reached (and equalled) the fixed timeout value. Although the piece of code met its specification, an anomaly was raised during the analysis which suggested that it would be safer if the code checked for the timer being equal to or greater than the timeout value. This was agreed by the manufacturer and the appropriate section of code was changed.

Overall, as a result of the analysis 12 code changes were made, 9 specification changes were made and no further action was taken on the remainder. Those for which no action were taken were either comments regarding possible improvements to the system/software or were comments where further clarification from the manufacturer, possibly relating to system level protection outside the scope of the software, was able to allay the concerns of the analyst.

7. DYNAMIC VERIFICATION

The dynamic verification activity was split into two parts, consisting of module testing and system testing. Because of the small size of the software, a separate integration testing phase was not applicable. Both sets of test were conducted using a hardware test rig which had the usual facilities available on typical in-circuit emulators.

For the module testing a set of tests were defined for each module. These consisted of both black box (functional) and white box (structural) testing. However, in this case, instead of using details of the source code as a means of determining the white box tests, the Semantic Analysis results were used instead.

The black box tests were initially defined using techniques of equivalence partitioning (mid-value) and boundary value analysis. Mid-value analysis essentially involves the choice of values for each input variable so that its input domain is covered. Boundary value analysis is based on the notion that if there are values where the value of a component changes, then errors of coding are likely to be made in the handling of those boundaries. This analysis procedure therefore chooses test data on or near those boundaries.

The test cases for the black box tests were chosen using these techniques from both the OOPS specification and the intermediate level natural language specification. Although the OOPS specification was at the system level, the small size of the overall software permitted module level test cases to be chosen from this document.

The selection of the test cases for white box testing was performed by using the Semantic Analysis results to ensure that all paths through each module were tested. This was feasible on this project because of the relatively small size of each of the modules and the small numbers of semantically

possible paths through each. Since the black box tests would cover a significant number of the paths through each module, the white box tests were chosen to supplement these and cover the remaining paths.

In more detail the way that the white box test cases was chosen was to select the input conditions defined by the predicate for each path in the Semantic Analysis. Consequently, for the example Semantic Analysis output shown in figure 1, path 1 would be exercised by setting the input variables such that conditions 1, 2 and 3 were false. The individual outputs would be checked on the rig to ensure that they agreed with those that had been both revealed by analysis and verified against the specifications.

In other cases, for example where the Semantic Analysis showed that there were loops or other break points in the code (for example lower level procedure calls), the results of the static analysis would be used to determine suitable breakpoints in the code at which execution should be halted for the inspection of intermediate results and the setting of new values. For some of this work a number of code inserts were necessary to act as test stubs to capture data.

A total of approximately 100 module tests were carried out, which gave complete path testing through each of the modules. A number of the tests failed due to anomalies that had been discovered during static analysis, however, no additional, previously unknown faults were found during the module testing.

The system tests were designed to investigate the behavioral aspects of the complete software and were intended to provide supporting evidence that the system satisfied its top level requirements. These tests were therefore classified as validation and black box tests and were chosen from the top level requirements specification rather than any of the lower level specifications or static analysis, except in the respect of timing.

As has been mentioned earlier, there had been concern, right from the derivation of the formal specification, about aspects of the timing, particularly relating to interpretation of the input signal. Although these problems had been investigated during the Compliance analysis, such issues are most easily investigated during dynamic testing. Consequently a set of tests was derived for investigating and quantifying this particular problem which was largely a module interface and integration problem.

The culmination of these tests into the timing issues was to confirm that the system was unable to meet a particular signal performance parameter that had been set in the original requirements. This was not considered to be a significant problem since, firstly, the performance figure defined had been chosen somewhat arbitrarily and, secondly, the effect was fail safe. However, since with a system of this sort being more fail-safe than intended has the consequence of causing destruction more often than is strictly necessary, there were potential cost issues to consider.

For this particular problem more detailed testing and investigation was able to identify why the particular problem was happening and to suggest a change in value of a number of software parameters to eliminate the problem.

8. CONCLUSIONS

The ultimate justification of the use of rigorous techniques, such as those described in this paper is whether the system is error-free in service. That has been shown to be the case on this project, although use of the system has been very limited to date. There is therefore no statistical evidence so far to show that the use of such techniques have been beneficial. Indeed it will never be possible to say with complete certainty that the same freedom from error could not have been justified using 'traditional' less rigorous techniques.

However, the use of the formal specification and the rigorous static analysis did reveal errors and deficiencies in the code which, without their correction, would have almost certainly resulted in erroneous operation of the system. Furthermore the use of such rigorous techniques has served to give a high level of confidence in the correctness of the software.

The aim of the work at the outset was to provide a comprehensive analysis by using disparate techniques. The use of a combination of static and dynamic activities has been shown to give a fair degree of overlapping, combined with maximal coverage of the problem domain. Where the techniques overlap there is a high degree of crosschecking, thus increasing confidence in the results. The goal of minimal effort has been achieved by employing the different capabilities of the techniques. In particular the extensive functional behaviour analysis afforded by static analysis combined with the dynamic timing analysis capabilities of testing has been shown to permit the investigation of all issues without enormous amounts of effort.

The work has confirmed the value of formality, at least on a project of this size where the operation of the whole system is at a level where it can be understood by a single individual without spending months of time learning and investigating the system. It has shown also that rigorous verification of the code against the formal specification is possible using Compliance Analysis techniques but it has illustrated that there are some non-rigorous, manual stages within this. Perhaps, because of this, the work has served to emphasise the likely difficulties of achieving complete proof of correctness on larger systems on which refinement over a number of levels is likely to be necessary.

Finally there is the issue of cost. Although the work has given as high a level of confidence in the correctness of the software as is considered possible, the costs have been relatively high in relation to the size of the software and the development costs of the code. However, it is considered that much of the cost is a consequence of the analysis being conducted post development and also, from having to be re-done following changes to the software.

It is considered inevitable that costs for the development and verification of safety critical software will be higher than those for non-critical code, although this is likely to be recouped in savings through increased reliability if the software has a large in-service life. Nevertheless, any additional cost for such software is likely to be tiny indeed compared to the costs of any potential software failures which may be expected without the initial expenditure on safety.

9. REFERENCES

1. IEC 65A(Secretariat)122, WG9. *Software for Computers in the Application of Industrial Safety-Related Systems.* Committee Draft, November 1991.

2. IEC 65A(Secretariat)123, WG10. *Functional Safety of Programmable Electronic Systems : Generic Aspects. Part 1 : General Requirements.* Committee Draft, May 1992.

3. IEC 880. *Software for Computers in the Safety Systems of Nuclear Power Stations.* International Electrotechnical Commission. 1986.

4. RTCA/DO-178A. *Software Considerations in Airborne Systems and Equipment Certification.* March 1985.

5. Interim Defence Standard 00-56. *Hazard analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* Ministry of Defence. April 1991.

6. Interim Defence Standard 00-55. *The Procurement of Safety Critical Software in Defence Equipment.* Ministry of Defence. April 1991.

7. *Object Oriented Process Specification.* S A Schumann, D H Pitt, and P J Byers. University of Surrey, Computing Science Technical Reports.

8. 00-31. *The Development of Safety Critical Software for Airborne Systems.* Ministry of Defence. 3rd July 1987.

Discussion

Question H. LE DOEUFF

A combien évaluez-vous le surcoût de production, de vérification, de validation d'un tel logiciel critique, avec la méthode que vous avez utilisée, par rapport à un logiciel simplement essentiel?

Reply

Using techniques such as formal methods, the early costs of the software development life-cycle will be higher than with "taditional" methods. However, savings should come during later stages of development (eg testing and maintenance). Wether these methods are cost-effective for non-critical code depends on the problems posed by software errors encountered in service and wether one is likely to encounter rework cost in any case as the user changes his mind as to what he wants the system to do.

Question C. BENJAMIN

Your effort was quite small. What efforts are you aware of are doing to automate the formal proof of the specification?

Reply

There are a number of individual projects going on in the UK and Europe to combine a proof system with a formal specification notation. Of most relevance is work undertaken through ESPRIT projects, for example on the RAISE toolset. In terms of MALPAS, we are continually enhancing the tool to improve the effectiveness of its proof ability (including the current development of a complete enw toolset), but we are not at present working to integrate the tool with any particular formal specification language.

Question R. SZYMANSKI

Will your approach work cost-effectively for large programs? (i.e. does the effort increase linearly or exponentially when the code size doubles?)

Reply

On this particular project, there were significant "end effect" costs because the system was so small. As the size of the software increases, we should expect initially to see economies of scale. However, beyond a certain size, one would reach a level of complexity where the difficulty of producing a formal specification would start to increase the costs.

Question L. COGLIANESE

To what degree did the fact that the program was written in assembler help or hinder your efforts?

Reply

Overall the fact that the code was assembler increased the formal verification effort since one had to refine the (high level) Z specification, written in terms of system level variales, down to the assembler level where one is dealing with specific assembler commands (such as bit matching and double length arithmetic). High level languages would have meant that one does not have to go down to such a low level.

Question W. ROYCE

In a retrospective way, when applying formal methods to existing code, are certain languages semantics (i.e. C, Ada, Fortran, Cobol, C++) better than other? For example, is Ada tasking easy or hard to verify?

Reply

Yes, certain languages are better than others. C, for example, is difficult because there are few disciplining features. In comparison, Ada would be better. However, Ada tasking is very difficult and must be excluded.

A DISCIPLINED APPROACH TO SOFTWARE TEST AND EVALUATION

BY
J. LEA GORDON
ASC/YFEA
WPAFB, OH 45433

Abstract

This paper discusses the impact DOD development standards and Integrated Product Teams have had on influencing F-22 cockpit Controls and Displays software test and evaluation.

Integrated Product Development Teams

Recently, The United States Air Force instituted a new approach to weapon system development which employs Integrated Product Development Teams (IPTs). In the past, a system was developed as a series of independent activities performed by different groups commencing with subsystem design and proceeding step by step to system deployment. At each step along the way, new groups of people began their activity where others left off. Communications between groups was practically nil. If the manufacturing group needed to understand a particular design subtlety they would have to seek out the assistance of the principle design engineer on an ad hoc basis who would then rely on his own resources to answer their questions. Whenever the principle design engineer became unavailable because he was reassigned or otherwise changed employment, the manufacturer was forced to make a best judgment and press on. Test engineers, support equipment designers and logisticians suffered similar problems. If they needed help in understanding a particular design, they would have to seek out the principle design engineer and hope he was available to assist them. Under the IPT concept, major segments of an overall weapon system are managed by a product team populated with both contractor and government personnel with a wide range of disciplines. The IPT approach to system development is a parallel process. The team manages the development of their product from the design phase through test, manufacturing and deployment. All disciplines including program management, financial management, contracts, data management, configuration management and logistics, as well as design, test and manufacturing engineering are represented on the IPT. All parties become involved with the system early in the requirements definition phase of the program and participate in each step of the decision making process throughout the development cycle. Experience gained on the U.S. Air Force F-22 program provides the basis for this paper which will illustrate how the IPT concept has improved the software design and test process.

Background

The use of software in complex military systems has grown rapidly in recent years. Expanded memory devices, faster processors, reduced form factors, economical power consumption and low prices have all led to the expanded use of microprocessors to implement system functions which were

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

formerly implemented in electronic hardware components. Discrete resistors, capacitors, operational amplifiers, logic gates and transistors have been replaced in many applications by miniaturized, programmable clocked sequential machines. The reason for this phenomenon is simple. Programmable devices are flexible; software is easier to change than hardware. Hardware change is labor intensive and requires a physical component replacement. Software can be changed with a key stroke. An advantage of hardware is that the design is visible through schematic and physical layout drawings. Changes can be easily identified through the drawing release system or if need be by an actual equipment configuration audit. Hardware change control is ridged and highly visible. On the other hand, software is not visible. Nothing about the software is revealed by an external examination of the processor it resides in. Moreover, the processor code listing provides little understanding of the design. As we will see in the following paragraph, many steps have been taken to document software development in order to provide design visibility and software change control.

Requirements

The United States Department of Defense formally recognized the necessity to design and document Mission Critical Computer Software (MCCS) for use in United States military systems systematically by mandating that it be developed in accordance with Department Of Defense Standards 2167 (DOD-STD-2167) and 2168 (DOD-STD-2168). These documents provide the framework for software design, development, test, documentation and quality assurance. They are sufficiently flexible to permit the tailoring of specifications to fit a variety of software applications. Regardless of the particular software effort to be developed, the first step in the requirements process is to generate a Systems Requirements Document (SRD). This document defines overall system requirements in terms of performance parameters. In addition, the SRD flows down pertinent software requirements called out by the two DOD standards cited above such as top-down design, structured programming, use of higher order programming languages, quality assurance measures and other specifications tailored to the application. Frequently, the initial version of the SRD is written by the procuring agency with assistance from his customer, the user. It may also be submitted to industry for review and comment. When possible the final version of the SRD is discussed with potential contractors in an open forum setting before it is released as part of a Request For Proposal for system development.

After contract award, the SRD becomes the responsibility of the prime contractor. It is used by the contractor as a basis to partition the total system into smaller segments or subsystems to perform delegated functions. Each segment or subsystem is described by a Prime Item Development Specification which comprises both hardware and software elements. Each distinct software element is called a Computer Software Configuration Item (CSCI) and is defined by a requirements document which contains software design requirements directly stated by the SRD plus additional derived requirements which result from system partitioning or from the refinement of particular performance specifications. All stated requirements must be traceable to system performance requirements stated in the SRD. The requirements document is a "design to" specification which establishes CSCI functional capabilities, performance values and tolerances, input/outputs, se-

quencing control, error detection and recovery, real time diagnostics, operational data recording, quality control provisions, operating limitations and other requirements peculiar to the software application. The requirements specification establishes the baseline for software design.

The actual software design is documented in a detailed specification which describes the CSCI structure, functions, languages, data base and smaller computer program components. It also describes the overall flow of both data and control signals within the CSCI, timing and sequencing of operations and other pertinent information. After the software has been coded and tested the design specification describes the final software product. The design specification for software is analogous to engineering drawings for hardware.

Software Design

Software design is a top-down process which results in the synthesis of transfer functions and algorithms to be hosted in a hierarchy of integrated program code building blocks also evolved in the design process. Top-down design facilitates bottom-up software testing and minimizes design changes by defining system interfaces before detailed algorithms are produced. The five major steps in the design process are listed below:

1. Define software design requirements.

2. Perform external system level interface design.

3. Lay out the software Architecture.

4. Perform module level interface design.

5. Synthesize module transfer functions and algorithms.

The first step in software design is to conduct a software requirements analysis based on the Systems Requirements Document and generate the computer software requirements specification. Where possible, requirements are stated in terms of the CSCI input/output interface signals and transfer functions.

The second step in the top-down design process is to address CSCI external interfaces. This entails designing software routines for devices which accept input timing, control and logic signals from sources outside the CSCI and provide output variables in the form of electrical signals to exterior destinations. Typically, the input/output devices interface with a data bus such as that described in Military Standard 1553B. External interface requirements are often established through an interface control working group. IPTs are accustomed to working in groups and are very effective in this forum. It is important to accomplish the external interface design before proceeding further to avoid signal characteristic incompatibilities and timing problems.

In step three, the designer evaluates how the overall software design can be broken down into smaller functional elements. The term software architecture refers to the type and arrangement of building blocks which are linked together to construct the CSCI. In some cases, software architecture is a constraint imposed on the system design by other factors such standardization. Barring such a constraint, the product software could be implemented in one large central processor or it could be distributed over several smaller processors. In either case, the designer partitions the CSCI into smaller elements called modules, components and

units. The smallest element of software is the unit which consists of about 200 executable lines of software code. Partitioning simplifies the programming task and facilitates tracking and documenting the software design.

In step four, the designer develops routines to interface smaller software modules together to produce the integrated CSCI function. Particular attention is given to module connectivity and the timing and sequencing of data.

The final step of the design process is to synthesize module transfer functions and algorithms and document this product in the detailed design specification. The existence of firm requirements eliminates costly redesign which can result from floating interface specifications.

Software Testing

Software testing is the process of executing a computer program with the intention of finding errors. Historically, the individual who designed a particular computer program or algorithm also coded the program and served as the tester of the resultant product. The problem with such an approach is that the designer/tester tends to delineate test procedures which verify that the program executes as coded. Verifying that code executes as expected in the laboratory is quite different from verifying that a system satisfies performance requirements under operating conditions. In order to accomplish the latter, software test requirements must be stated in terms of system performance requirements which can ultimately be verified by test at the system level.

F-22 Controls And Displays Test and Evaluation

Development of the United States Air Force F-22 aircraft cockpit Controls and Displays (C/D) is managed by an Integrated Product Team. The team is concerned with every phase of product development and operation from "cradle to grave" including performance, cost, schedule, testability, producibility, and supportability. Team membership is made up of both contractor and government personnel representing all disciplines involved in the business of systems acquisition. The F-22 System Program Office is committed to the premise that requirements definition and software documentation are vital to successful system development. Experience has also demonstrated that rigorous testing is a mainstay in assuring proper software design. With these factors in mind, the team approach to software design and test has adopted the following guidelines:

A. Start with requirements and adhere to established software development rules.

B. Integrate software test into the systems engineering process.

C. Rigorously test software to system performance requirements.

From the foregoing discussion on requirements, it is evident that adequate direction is on the books to insure that military software development is properly conducted and documented. Unfortunately, in the past it has been difficult to implement existing direction. Prior to the use of IPTs, the task of writing software design requirements was left up to the designer. Eager to get started, the designer would often times forego conducting and documenting a software requirements analysis. Instead, he would code and test bits and pieces of functions which he thought he understood. He would iterate

this procedure until he finally came up with some conglomeration of functions that seemed to work. The code for this conglomeration would then become the software "design". In one notable case, a major avionics system actually entered flight test without having either an approved requirements document or a detailed design specification. The order of the day was to flight test the system, analyze the results and fix anything that didn't work correctly. With no documented requirements, it was nearly impossible to determine what worked and what didn't. Needless to say, that project turned out to be a disaster in terms of cost overruns and generally poor system performance. The use of IPTs has alleviated this situation by enforcing standard software development practices. IPTs endeavor to state software performance requirements in clear, concise, unambiguous and quantitative language so that all parties understand what the software is supposed to do and why it is suppose to do it that way.

Probably the most challenging job of the detail designer is coping with changing requirements. IPTs are very useful in stabilizing design requirements and resolving conflicts. It is worth noting that requirements sometimes must change for good reason. For example, an integrated logistics support requirement might be stated as follows: "The product software shall provide a post mission in-flight built-in-test diagnostic capability which shall fault isolate equipment malfunctions to the line replaceable unit level prior to aircraft landing." The intent of the requirement is to increase sortie rate by eliminating the use of special ground support equipment to trouble-shoot failures occurring during a mission. While this requirement may be desirable, it's implementation necessitates the addition of special avionics built-in-test equipment which may unduly burden the system with additional weight, power consumption, and heat dissipation demands. A cost/benefits trade analysis would be necessary to resolve this issue. IPTs have proven to be particularly useful at evaluating alternative solutions to such conflicting requirements. In this case Logisticians and Mission Planners understand the operational need for built-in-test fault isolation. Designers understand the extent and complexity of the hardware and software needed to implement the required capability and the time required to accomplish the task. Program management understands budgetary limitations and schedule constraints. Considering all elements of the problem; performance, cost and schedule, the IPT is eminently qualified to decide the fate of the stated requirement.

In the case of the F-22 Controls and Displays, software development is being executed in strict compliance with DOD standards. Software design did not begin until after the Software Requirements Specification (SRS) and the Interface Requirements specification (IRS) were approved at preliminary design review. Coding of the design will not begin until after the Software Detail Design Document (SDDD) and the Interface Description Document (IDD) receive approval at critical design review.

Furthermore, software test has been an integral part of the systems engineering process from day one. Test planning was initiated in parallel with software requirements definition. Each requirement in the software requirements specification is identified by number in a separate paragraph. Numbering enables a requirement to be traced to the Software Detail Design Document and to the Test Verification Matrix. It also controls the addition of extraneous requirements not derived from the original SRD and eliminates what has been called "creeping elegance".

The IPT test engineer is responsible for knowing how the total system is supposed to perform at the operational level. He participated in translating operational characteristics into stated software performance requirements. Once overall requirements were defined, the C/D IPT test engineer helped generate a comprehensive system Test and Evaluation Master Plan (TEMP). The TEMP identifies the entire test process from the checkout of individual components to modules to subsystems to system segments and finally to the integrated system.

Using the TEMP as a guide, the test engineer formulated detailed test procedures for the C/D product in parallel with the design process. He concentrated on developing test requirements and test cases which are directly traceable to system performance requirements. Software test cases were established without regard to code. The following is an example of test requirements stated in terms of quantifiable software input/output parameters and the CSCI transfer function:

(1) The nose index Uncage Command (UC) output signal shall be set to 5.0vdc plus or minus 1.0vdc when BOTH of the following input conditions are present:

A. Lock-On signal (LO) equals 5.0vdc plus or minus 1.0vdc

B. System Ready signal (SR) equals 5.0vdc plus or minus 1.0vdc

(2) If either the Lock-On signal or the System Ready signal is 0.0vdc plus or minus 1.0vdc, then the Uncage Command output shall be set to 0.0vdc plus or minus 1.0vdc.

In this example, two requirements have been stated. First, the logic transfer function of the software unit is specified to be:

UC=LO AND SR

Secondly, interface signal logic levels are defined:

Logical one equals 5.0vdc plus or minus 1.0vdc

Logical zero equals 1.0vdc plus or minus 1.0vdc

The requirements stated above represent functions which are "testable" independent of the code used to implement the software. The following are examples of requirements which do not relate to system performance and which are therefore not "testable".

1. The Software shall generate the nose index marker as necessary.

2. The Software shall satisfy all system constraints.

3. The Software shall be sufficient to support mission planning.

4. The Software shall self test the processor

The IPT test engineer also writes the Test Description Documents and the Test Procedure Documents which are used to execute software Preliminary Qualification Testing (PQT), Formal Qualification Testing (FQT) and Acceptance Testing (AT). Software testing is performed in a "bottoms-up" manner at the unit, component and module level. PQT is the most stringent and stressful type of testing that is performed to assure that the software satisfies transfer function design requirements. PQT verifies that all legitimate combinations of operational input signals to a particular software element produce the expected outputs signals. It also verifies that other input signal combinations which are not expected to occur during

normal operations do not cause the software to "lock-up" or cause spurious outputs or other "glitches". PQT also introduces boundary value conditions which exercise the software tolerance limits under worst case conditions in an effort to produce faults. FQT is conducted at the CSCI level and verifies that application of the proper input signals to a CSCI produce the proper output signals. Acceptance testing is not a software design verification test. It is a functional test which demonstrates that hardware and software are operating to nominal performance levels. IPTs strengthen the software test and evaluation process because test is treated as an independent discipline which begins with the origination of system performance requirements rather than as a follow-on effort to software design.

Conclusions

Existing DOD standards are in place which adequately describe a viable software development process. Implementation of these standards coupled with a rigorous test methodology will result in high performance, cost effective software.

Integrated Product Teams facilitate communications, serve as corporate memory and provide continuity to the system development process. IPTs provide the management muscle necessary to enforce existing DOD software development standards and produce a quality product.

Discussion

Question L. HOEBEL

Could you describe the 10% of F-22 software that is not in Ada? What language, what functionality and why not Ada?

Reply

9% is in assembly. The assembly is done at Hughes Aircraft. It is needed because of the timing of the chip in our Central Interfce Processor (CIP).

1% is in C because it is more cost effect to use the language in off-the-shelf equipment than to rewrite in Ada.

THE UNICON APPROACH TO THROUGH-LIFE SUPPORT AND EVOLUTION OF SOFTWARE-INTENSIVE SYSTEMS

Donald Nairn
Sonar Signal and Data Processing Division
Defence Research Agency
Portland
Dorset
DT5 2JS
UK

1. SUMMARY

A new approach is presented to the through-life support and evolution of software-intensive systems, involving a sequence of development contractors. The problems addressed are to do with the avoidance of a "through-life dependency" on any of the development contractors, and of achieving a unified engineering system description without imposing undue restrictions on the use of evolving software methodologies.

The UNICON approach is to vest the through-life continuity in a unified engineering description held in a new type of Project Support Database. This scheme now views the development contractors as having a "jobbing" role.

The technical feasibility of this approach was held to depend on achieving a much more complete engineering description - from the Statement of Requirement, through to the mapping of the software onto the hardware (variants) - than any yet available. This led to the development of the UNICON system description language, based on extensions to the familiar ENTITY/RELATION/ATTRIBUTE notation.

Details are presented of a reverse engineering experiment which captured a description of an existing 27-processor equipment within a prototype UNICON database. This also involved codifying the software production processes, the results being verified by replicating an existing software build, purely from the UNICON information. An outline is given of the UNICON Project Support Environment.

A new approach is proposed to the standardisation of Software Support Environments, whose objective is to support the transfer of work between conforming Environments, rather than to unify their implementations. This idea is contrasted with the more familiar Public Tool Interface standardisation scheme.

2 INTRODUCTION

Over the past five years the UNICON research programme has produced a new approach to the through-life support of large software intensive systems. With the latest trends towards system integration, there is an increasing awareness by Projects of the need to avoid a through-life-dependency on any one development contractor. Thus on-going system evolution, perhaps over a thirty year span, may have to involve a sequence of development contractors - each operating on a "jobbing" basis.

For such an approach to be feasible, it would require a standard of Engineering Documentation well beyond anything yet demonstrated, since system design-continuity would now have to be supplied by this documentation. Furthermore a unified engineering notation would be needed for the entire system, capable of describing different design methodologies, and of evolving to encompass future advances.

Although aimed initially at capturing a unified description of the work done by many development contractors, perhaps through a process akin to "reverse-engineering", it can readily be appreciated that such an engineering notation could also be used to co-ordinate the development work itself. From this it follows that a UNICON Software Environment is equally applicable to both system support and to software development. (In fact UNICON has been specifically designed to co-ordinate very large development teams - perhaps one thousand engineers or more).

Finally the notion of basing a new standard for Software Support Environments on an Engineering Description Notation is contrasted with the present Public Tools Interface (PTI) approach - as in the PCTE programme etc. The question is raised as to whether the capability of transferring development work between conforming Environments is a much more direct standardisation objective than some capability to implement individual Environments from standardised tool-components.

3 THE UNICON APPROACH

3.1 Fig 1 The Through-life Support Problem

Current practice is for a new software issue to be made up from separate components, each compiled by the original Development Contractor.

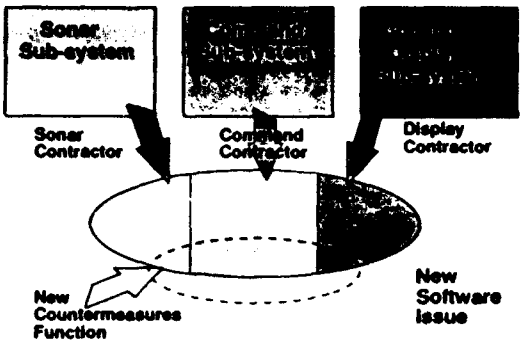
-- Elaborate interface specifications are used to co-ordinate the software builds from different Contractors.

This leads to a through-life dependency on the original development contractors.

Views expressed are those of the author

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

Traditional approach to through-life software management:-



***Through-life dependency upon 1st wave Development Contractors**
***How do you go about new Counter measures Function?**

Fig 1

-- Perhaps fifty years in some cases.

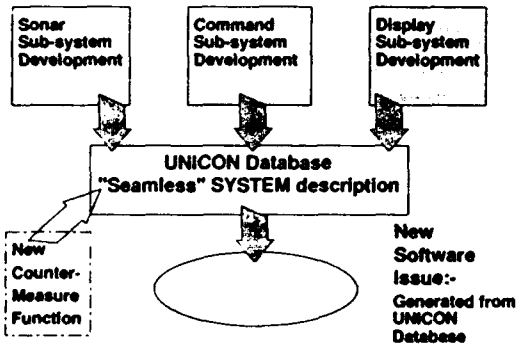
How do we cope with an evolutionary requirement which spans a number of development areas ?

-- Such difficulties will eventually be the limiting factor in upgrading existing systems.

-- These problems will become more acute with the moves towards integrated systems.

3.2 Fig 2 The Unicon Solution To Through-life Support

New Approach To Through Life Software & Design Management



***No through-life dependency on any Contractor**
***Uniform & "Seamless" description held for entire system**
- Updates eg; Countermeasures Fn specified using UNICON information.

Fig 2

The UNICON database captures a description of each Development Contractor's work, to form a unified "seamless" system description.

-- This may involve "reverse-engineering" each Contractor's documentation.

Complete software issues are now generated from the UNICON database - probably by a Systems-Support Contractor.

-- There is no dependency on any of the Development contractors.

-- Thus the UNICON systems description must include a codification of the software production processes (ie compiling, linking etc).

Requirements for evolutionary enhancements, are now specified in terms of the database system description.

-- The Systems-Support Contractor is involved in elaborating this requirement - and eventually in overseeing the upgrade of the database. They would have to certify that they can now support the new function, before the development contractor can be paid off.

3.3 Fig 3 Unicon Describes Systems Using An Extended ENTITY/RELATION Model

System description in UNICON is based on an extension of the familiar ENTITY/RELATION notation. Fig 3 shows that a UNICON ENTITY is just an Identifier, which can enclose a nested set of child ENTITIES (ie child-Identifiers).

-- UNICON ENTITY Identifiers are in fact unique database reference numbers (rather like "path" names). A User Name for an ENTITY is optional, and is treated as an alias.

RELATIONS express "links" between ENTITIES. Sets of RELATIONS are held in parallel planes - rather like a multilayer electronic circuit board, where the ENTITIES correspond to the microchips, and the sets of RELATIONS correspond to the separate layers of inter-connections.

-- Separate planes of RELATIONS express separate classes of "links" - eg links which describe connections between code-module ENTITIES, are of a different class to those links which assign code modules to processor address-spaces.

The bulk of the information within a UNICON database is held as "anonymous" ATTRIBUTES, which are bound to individual ENTITIES.

-- It is rather like each ENTITY/ RELATION providing extensive management and version-control facilities for a "safe-deposit" box, without taking any interest in the contents held within each box - other than the specification of the links to other boxes.

3.4 Fig 4 A Typical Unicon Project Support Database

Fig 4 shows that a typical UNICON database is configured rather like a three-dimensional stack of bricks.

-- Each "brick" is itself made up of numerous ENTITY/RELATION planes.

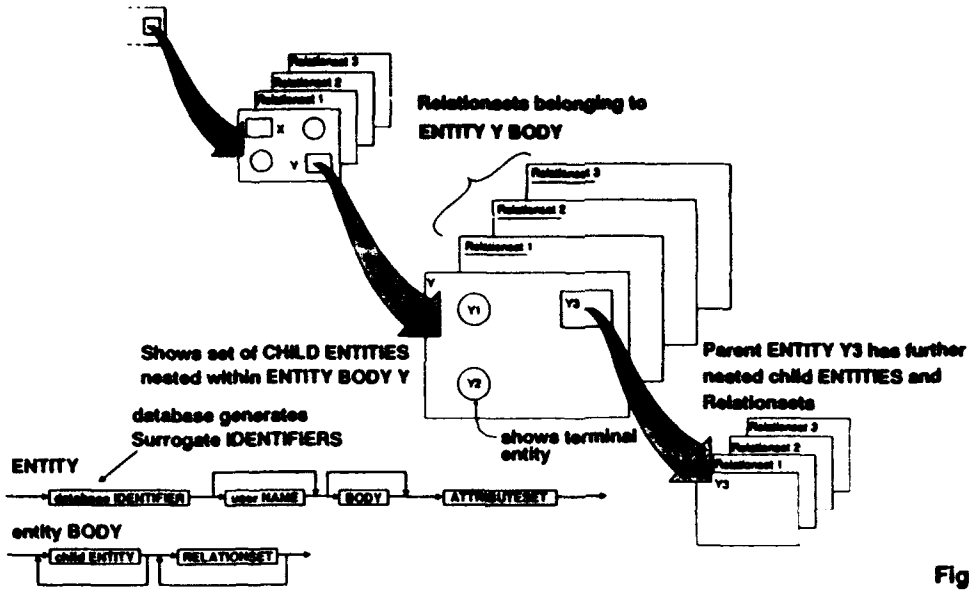


Fig 3

The Database x-axis holds parallel sets of hierarchies. These hierarchies model the system documentation - from the Statement-of-Requirement, through to the mapping of the software modules onto the processor hardware, and then on to the hardware installation diagrams etc.

-- There exists a one-to-one correspondence between the database ENTITIES and those of the system documentation - thus the UNICON database has an object-orientated configuration.

-- Since UNICON constructs an ENTITY/RELATION schema-model of the engineering documentation, it preserves the locality properties of the system information - thus very large systems need not incur significant access-time overheads, even with modest PC workstations.

The Database y-axis holds system design-variants - one for each platform. Each variant is held in the form of an incremental "overlay" operator, so that any one variant can be generated as required, by operating on the base design.

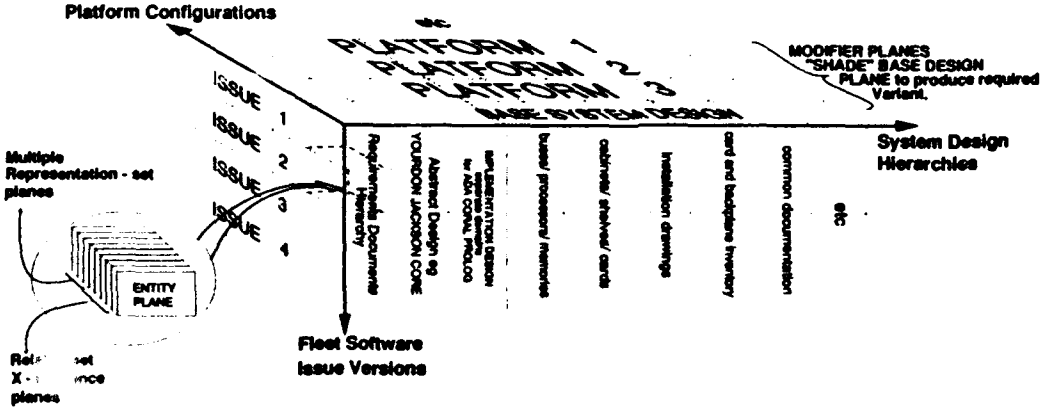
-- Apart from being more efficient in storage, this supports direct comparisons between platform variants.

The Database z-axis shows the sequence of software issues, for the base-design and for all of the platform variants.

As with platform variants, UNICON constructs each version "on-demand", from a flat library of common modules - thus the same library modules can be re-used for the construction of many versions. Similarities/differences between versions can readily be found.

A key design concept is that UNICON holds information in one place only - all other uses are provided by references

-- Thus processor memory definitions held in the



UNICON DATABASE MODELS WEAPONS SYSTEM STRUCTURE

Fig 4

hardware inventory, together with its shelf location and customising-backplane information, are an integral part of the code-generation process.

3.5 Fig 5 Unicon Database Has A Graphical User-interface

UNICON has a WINDOWS type graphical User Interface - rather like a simple Computer Aided Drawing utility, as in Fig 5.

-- However in "drawing" Fig 5, the actual process (transparent to the User) was to declare an ENTITY/RELATION network to the database, and then to have the database display its contents using a particular set of ICON Representations.

UNICON has an extensive set of Navigation and Trace facilities to support User database queries via the screen.

-- These are to be supplemented by a small library of navigational functions written in C, which may well allow SQL-type query interfaces to be constructed if required.

The present UNICON implementation employs an event-driven configuration which is compatible with virtually all of the present graphics standards - eg WINDOWS3, X-WINDOWS, etc.

-- Since graphical representations are held in absolute co-ordinates within the database - using the same grid as POSTSCRIPT - simple handlers can therefore map UNICON pictures to any screen resolution.

3.6 Fig 6 Multiple Representations Provide Alternative Database Views

Fig 6 shows how UNICON can form a new View into the database, by elaborating sub-systems to any depth, starting from any chosen diagram.

-- This operation involves a non-linear transformation, since the amount of "space" on the new composite diagram, claimed by any one sub-system, depends on the content and nested elaborations of that sub-system.

-- The transformation to form the new composite View is automatic - although the User may wish to pretty up the new lay-out, the information will always be accurate.

This new composite View can be used for database interaction - eg for system design - there being no functional distinction between single and composite Views

-- Alterations on one View will of course change the other Views, since they are all generated from the same underlying data - this may require some prettying-up of associated (off-screen) Views.

Any number of composite Views can be held as parallel Representations bound to the ENTITY frame describing any one diagram.

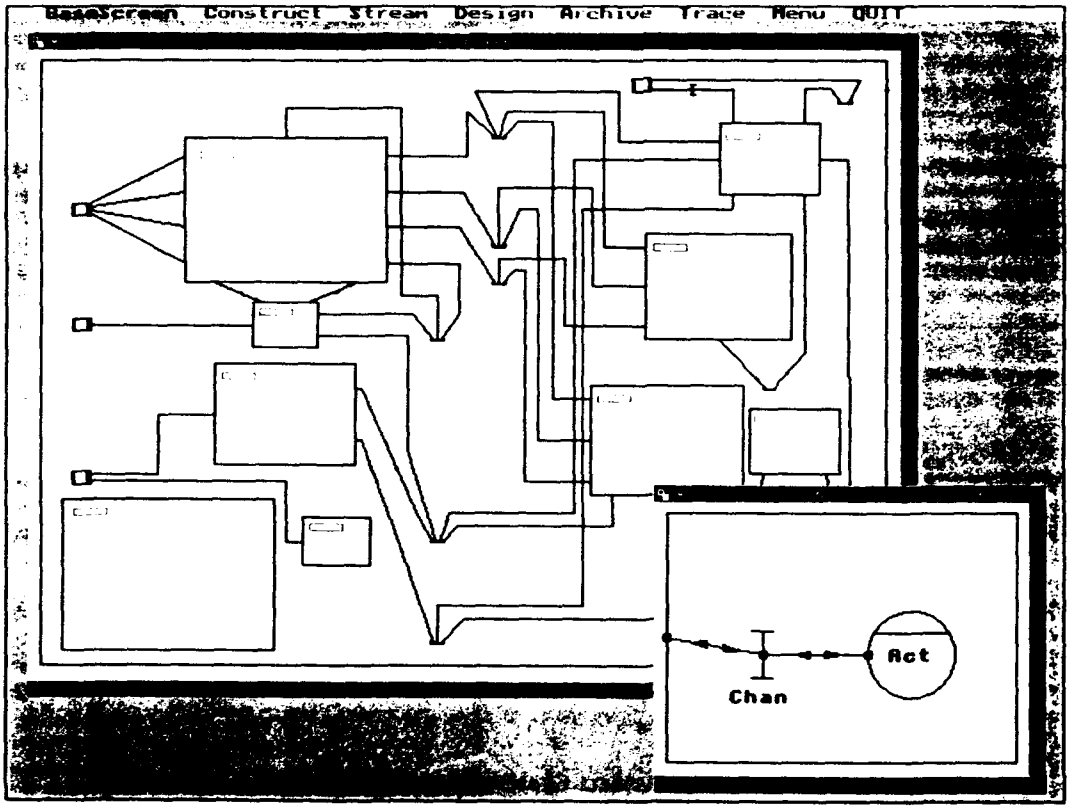


Fig 5

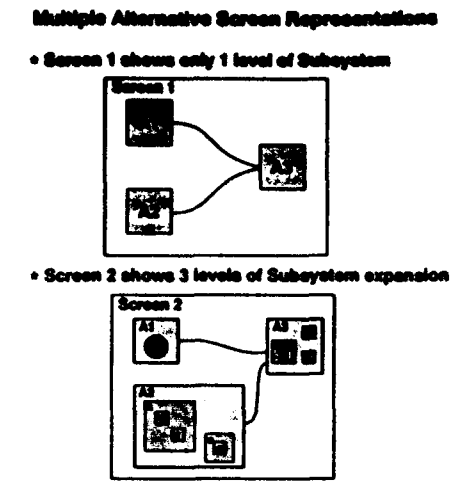


Fig 6

-- An existing composite View can also be used as a component within a superior composite View.

-- In this way it would be possible to perform a bottom-up construction of a single composite View for an enormous system - and print it out as POSTSCRIPT tiles to cover the wall.

The composite View facility provides an answer to the so-called "white space problem" - where individual diagrams in a hierarchy do not show enough context information, ie each diagram is surrounded by a sea of white space.

-- ADA Bhur diagrams are particularly prone to this condition.

3.7 Fig 7 Illustrating How Unicon Documentation Is An Integral Part Of The Code Construction Process

Fig 7 provides a somewhat loose illustration of how the UNICON ENTITY/RELATION notation guarantees the accuracy of the design documentation, by making each diagram an integral part of the code-construction process.

-- As shown, the RELATIONS between the top-level sub-systems can be regarded as "ducts" which will subsequently carry inferior RELATIONS generated by lower-level sub-systems, and eventually by code modules.

-- The whole process is analogous to running "wires" between code modules, within a system of conduits provided by the superior sub-system links.

-- Note that an interface can be inspected by opening a "duct" and tracing the "wires" - in effect a new type of database query facility.

Thus it is seen that UNICON supports a process of top-down design, followed by bottom-up elaboration.

It is possible to regard the UNICON ENTITY /RELATION notation as a meta-language which sits above conventional languages such as ADA etc. In fact UNICON provides a transformation, closely analogous to a conventional compiling process, in which a hierarchy of sub-systems is transformed to become a very large "flat" network containing only code modules - which can now be fed to one or more compilers.

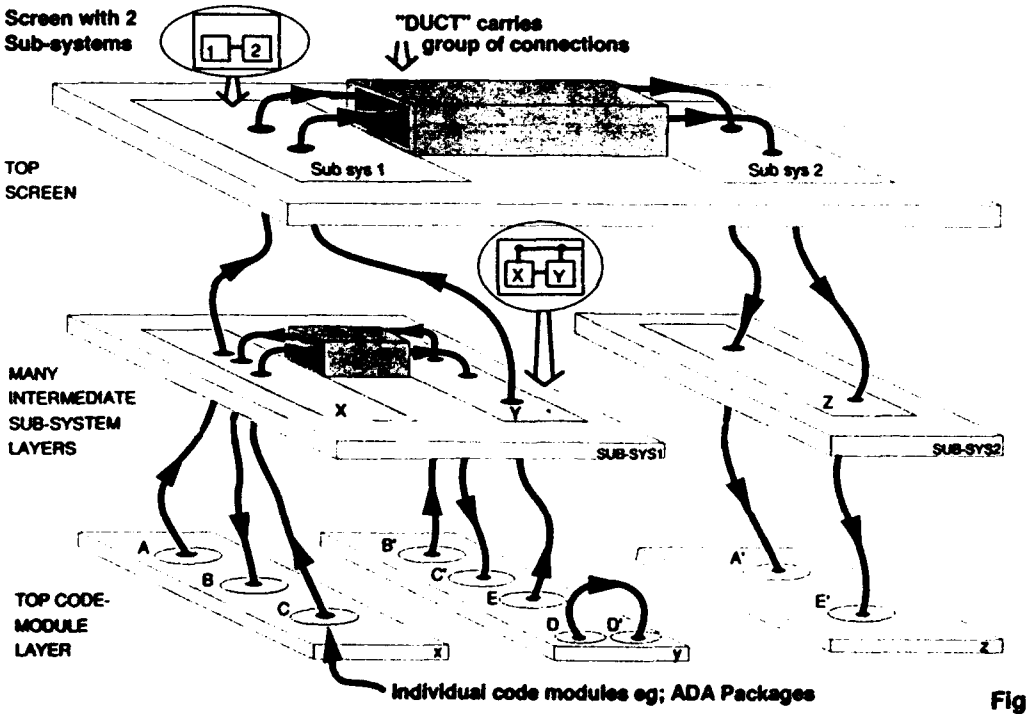


Fig 7

-- It is noted that such facilities provide new options to the system designer - it is now open to choose between having x code modules at sub-system level k, or having 100*x simpler code modules at the lower level of n say, without in any way reducing the degree of checking done by the compilation process(es). (Ref 2 gives a good treatment of the continuing need for a sub-system construct to supplement ADA).

It is evident that the above "compiling" transformation involves all of the diagrams in the hierarchy as an integral part of the code-generation process - thus guaranteeing the consistency of all levels of the documentation.

-- If say the top-level diagram were to be deleted from the database, UNICON would not be able to trace connection paths whose threads passed through the deleted RELATIONS.

Continuing with the meta-language interpretation, it would in fact be possible to express the ENTITY/RELATION description of the sub-system levels, as an explicit ASCII "source text". The UNICON transformation would now become a true symbolic compiling operation, yielding a large network of conventional (multi-lingual ?) source text modules, suitable for feeding into conventional compiler(s).

-- Such a symbolic approach would probably require the equivalent of the UNICON database surrogate Identifier reference system, in order to manage the large global name-space, used to express the connectivity of the resulting network within the source texts.

Finally, it is put forward as a conjecture, that the above sub-system "compiling" transformation may be an enabling technology to support a new approach to software-proving, where the above reduction process would be continued down to arrive at sufficiently simple code modules, that they could be regarded as true in themselves.

3.8 Fig 8 The Unicon Reverse-engineering Experiment

A Reverse Engineering experiment was undertaken, both to develop and to demonstrate the UNICON concepts. The aim was to capture a engineering description of an existing in-service equipment, within a UNICON database.

-- The equipment chosen has 27 processors configured on 12 buses - it was programmed in CORAL/MASCOT.

-- However, the actual experiment was undertaken on a pilot scale - a single shelf having 3 processors on one bus.

As shown in Fig 8, the MASCOT design diagrams were transferred into the experimental UNICON Environment by drawing same on the screen.

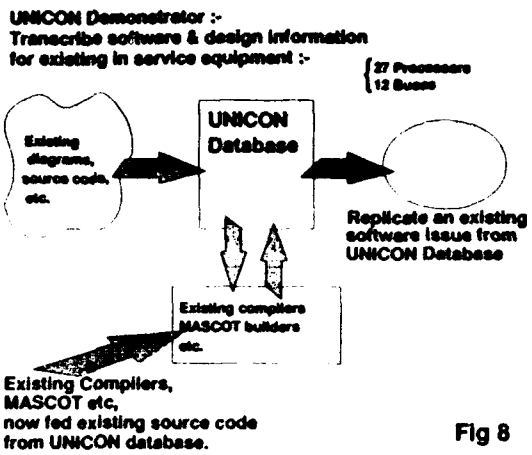


Fig 8

The aim was to prove the accuracy of the UNICON engineering description, by replicating an existing software issue - purely from the information held within the database, together with the original compilers etc

-- Thus it is noted that the reverse engineering task also involved codifying the proprietary software production processes, used in-house by the original contractor.

3.9 Fig 9 Experiment Used Code-generating Handlers

Fig 9 shows the overall scheme. The original software production was done using a sequence of compiling, composing, linking, etc utilities on a VAX mainframe. These operations were directed by a set of hand-written Batch Command files.

-- Thus the reverse-engineering task reduced to one of producing automatically an identical set of Batch Command files - using only the information present within the UNICON database.

The job was done in two parts. The UNICON database generates a separate ASCII Form file for each code module, which contains all of the information known about that module - the location of the source-text files, its connections to other module Forms, its processor memory location, etc etc.

The original contractor produced an application-specific handler, which read these ASCII Forms, and transformed the information into the required Batch Command file format

-- In effect, the handler aped the manual work of the original designers - only this time the database information which produced the design diagrams, was an integral part of the code-production process - thus ensuring the accuracy of the documentation.

It should come as no surprise that a significant amount of the reverse engineering effort was devoted to achieving a consistent set of diagrams - since this was the first time that any method was available for proving the original documentation.

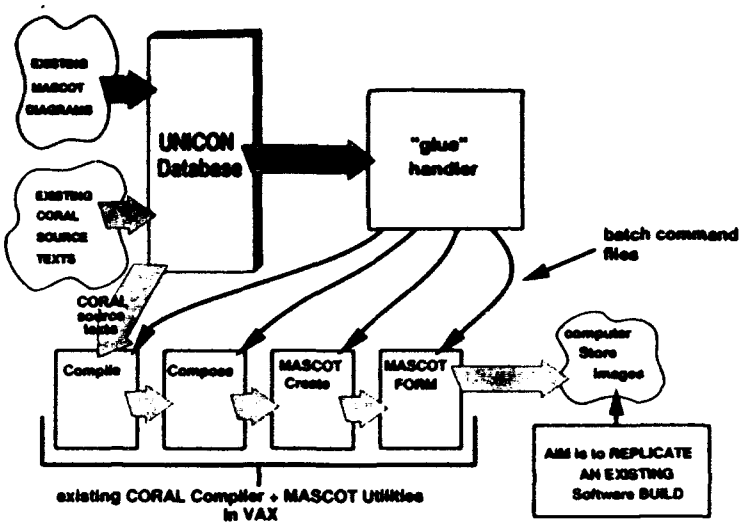


Fig 9

-- There is no implied criticism made here of the contractor - the situation is held to be an inevitable consequence of having to rely on manual checking procedures.

-- This result also provides strong evidence for the existence of substantial hidden costs involved in updating equipments, due to in-accurate documentation - even when used by the original development team.

At the conclusion of the experiment, the original Design Team agreed that the scheme could readily be scaled up to describe the whole equipment, and that techniques used were not specific to that particular job.

-- However, it has also to be recognised that only a minority sub-set of the UNICON concepts were exercised during this experiment.

3.10 Fig 10 Outline Of The Present Unicon Implementation

Fig 10 shows how the present UNICON implementation is structured as a Transaction Processor, a Database Manager, a Representation Manager, a View Manager and a User Interface Manager.

Database ENTITY/RELATION diagrams are drawn on the screen by passing object-streams to the Representation Manager, which expresses same in terms of graphics strokes read from the application-specific ICON store.

A Windows-type User Interface is provided, employing an event-driven configuration which is compatible with the main graphics standards - WINDOWS3, X-WINDOWS etc.

-- The transformation from absolute (POSTSCRIPT compatible) co-ordinates to pixel co-ordinates takes place within the User Interface - thus ensuring the portability of the database Representations.

The View Manager allows the user to look simultaneously at a number of pictures in separate WINDOWS. However the rest of the Environment is "aware" only of the single active picture.

The present implementation provides a good real-time performance on a 386 PC, and has some 30k lines of TURBO PASCAL code. A fully featured UNICON Environment is expected to occupy about 50k lines of code.

-- The very small code size reflects the simplicity of the underlying ENTITY/RELATION notation, and the use of a small set of tightly-coupled generic tools (eg the one tree-diagramming utility is used to manage the sub-system, the versions, the database segment, and the User-defined schema hierarchies).

-- The portability between UNICON implementations, of the large set of loosely-coupled (high-value) tools - eg compilers, word processors, code-analysers, project-management tools etc - is achieved through the use of simple Handlers, many of them being application specific.

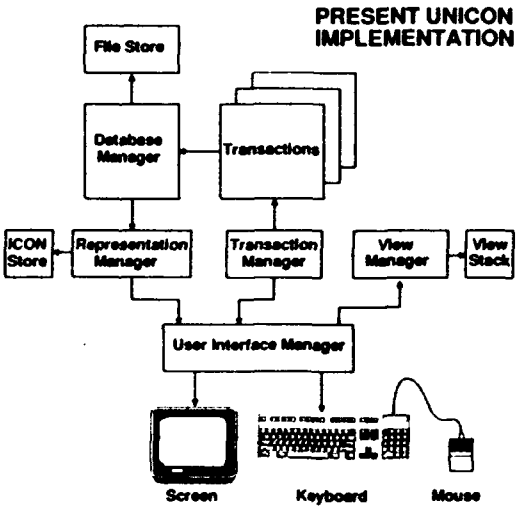


Fig 10

4. CONFORMANCE SPECIFICATION FOR A UNICON STANDARD

It is proposed that the only conformance requirement for any UNICON Environment, would be the capability to import work expressed in the UNICON ASCII Transfer Format - and the capability to re-export same (presumably after enhancement) using the same Format.

-- In particular the proposed UNICON Standard does not seek to specify the facilities provided by any one UNICON implementation - although the ability to generate the UNICON Format does imply the use of a small number of standard transformation algorithms.

-- Hence the actual facilities present within any one UNICON implementation will be a matter for negotiation between the User and the Supplier.

Since the Transfer Format consists of a relatively small number of ASCII stream types, each with a tight specification, it follows that there need be no restriction on how any conforming UNICON Environment is implemented - eg PASCAL on a PC, ADA on a SUN/VAX etc, etc.

Because the Transfer Format consists of ASCII record structure images, it follows that it is possible to import/export application information in the form of "anonymous" database records. Thus the UNICON Import/Export operations are at the level of trees of design-Versions, rather than at the ENTITY/ RELATION notation level.

-- The situation is somewhat akin to doing anonymous sector-by-sector disk-transfers, rather than reading and transmitting individual files.

A particularly attractive feature of this Transfer Format approach to conformance, is that individual Groups/Users can be allowed a considerable freedom to tailor their own Environments, without having to worry about issues of re-validation.

-- In effect, UNICON conformance testing is done "on-line", at each Transfer operation between UNICON Environments.

-- In practice such tailoring would probably be confined to the provision of alternative tools, so that the object - encapsulation properties of the Database Manager would also exert a powerful conformance-preserving influence.

5. SUPPORT FOR VERY LARGE DEVELOPMENT TEAMS

UNICON seeks to support large development teams through the database being structured as a tree of independently managed SEGMENTS.

-- Since any ENTITY/RELATION tree within a SEGMENT has a parent ENTITY within its superior SEGMENT, it follows that the arrangement is functionally equivalent to a single "seamless" ENTITY/RELATION tree-structure within a single SEGMENT.

A SEGMENT has its own autonomous Version Manager, which describes a tree of Versions within the SEGMENT.

-- Each Version in this tree records a "state" for the entire SEGMENT as a whole. A state is generated by an iterative indexing operation, which operates upon a flat library of ENTITY/RELATION objects, and on the nested ATTRIBUTE Libraries. Most library objects are common to many versions.

A SEGMENT also has a separate Configuration Version Manager, whose function is to select both a particular Version Number for the current SEGMENT, and to specify the configuration of the child SEGMENTS - and their respective Configuration Version Numbers.

-- In this way every Version Number in the Configuration tree, specifies both a configuration and a state for the present and all inferior SEGMENTS. (Thus each Configuration version can be regarded as a particular "view" of some part of the database).

-- In the case of the top SEGMENT, a configuration and a state for the entire database is specified for each Configuration Version Number - essentially through a cascade of index objects, distributed over the child SEGMENTS.

The separation of the SEGMENT and the Configuration Version-Managers, means that changes within any SEGMENT can be done in isolation from the rest of the database.

-- In general there can be many (inconsistent ?) SEGMENT-versions interpolated between those which are made visible to the rest of the database, through being called up as part of some new Configuration-Version.

A Section Leader can specify work-packages in the form of a particular configuration of SEGMENTS, which can be regarded as an independent and consistent sub-Environment

-- This could be implemented within a mainframe/network system as a side-branch of a Version-tree, or be exported to a stand-alone PC/SUN workstation using the ASCII Format, to start a new Version-tree - their being no functional distinction between local and remotely sited sub-environments.

-- Library objects are never changed in-situ - changed objects are held in working store, and become new library objects when a new Version is generated.

-- On completion of (a version of) the work-package, the information would be re-Imported

to become a child Version of the original.

- It would be the Section Leader's function to assimilate one or more of these child-Version work-packages, to become a new composite Version on a different branch of the tree, perhaps for export to the superior level - thus avoiding the classic "lost update" situation.

- It is noted that the above scheme exploits the UNICON state-based Version facilities, to avoid the necessity for checking-out or "locking" information while it is being worked on by more than one party.

In summary, it is seen that state-based Versioning is inherently capable of supporting database distribution - a branch of a Version tree can be exported to a sub-environment where it becomes the root of a new tree. A subsequent operation can re-import the entire tree, to become a side-branch of the original parent - in effect taking delivery of both the enhanced product, and of the (suitably pruned) "lab book" for audit trail purposes.

Besides managing its own object libraries, each SEGMENT issues its own (persistent) surrogate database Identifiers for ENTITIES/RELATIONS.

- The path-name or "stem" of SEGMENT surrogate Identifiers, passes through the Configuration of the superior SEGMENTS, this being a much more stable route which is independent of the actual contents of these SEGMENTS.

- The persistence properties required of surrogate identifiers, mean that they have to be preserved without change through all database versions - and through all Import/Export operations. This is achieved by the UNICON database using separate orthogonal coordinate systems, for the surrogate Identifiers and for Version management.

6. COMPARISON WITH THE PUBLIC-TOOL-INTERFACE APPROACH TO STANDARDISATION

The European ECMA PCTE programme, and the US CAIS programme, are both based on the concept of a Public Tools Interface (PTI) as a means of Environment standardisation. The notion is to enable Environments to be constructed through a mix-and-match selection of tools from different vendors.

- The idea is to create a market for the supply of tools which will be able to inter-work.

- A key objective is to achieve "open" evolutionary Environments which will not be dependent on any one supplier.

However it has to be borne in mind that the definition of a tool-coordination interface is a means to an end - the activity does not address the actual problems of producing adequate engineering descriptions and managing their development.

- It can therefore be noted that a standardisation approach based on finding a sufficiently general engineering description notation, would at least have the virtue of focusing attention on some of the actual problems to be solved - as has happened in UNICON - rather than on some perceived, but remote, enabling technology.

Ref 1 has noted that despite spending "astounding" amounts of money over a ten year period, large standardisation initiatives such as PCTE and CAIS have yet to be adopted by the software development community on any scale.

- They go on state that the basic feasibility of a PTI has yet to be proved, there being a substantial body of opinion that questions whether it is yet possible to define a PTI.

- This Reference also notes a growing acceptance that only a core of functions within an Environment have to be tightly integrated, with application-specific handlers being a perfectly adequate means of interfacing many loosely-coupled tools.

It is worth considering some of the implied assumptions behind the notion of a Public Tools Interface - that separately designed tools from different vendors can mesh together closely enough to implement the core of an Environment, and that these core-tools are of a sufficient value/complexity to be worth codifying a general interface definition.

- It is argued that empirical results from the UNICON programme undermine both of the above implied assumptions:-

- In UNICON many of the core tools merge together to such an extent as to blur their very identities - eg there is no graphics editor, since this function is incorporated within the (separate) database input and the database graphics output facilities. Also, Version management is meshed so closely with database segmentation and with Import/Export functions, much of it having to meet demanding real-time User-response constraints, that it is inconceivable that separate vendors could have been involved in any one implementation.

- On the other hand, the UNICON work demonstrates that, with a simple notation of sufficient scope, the core of an Environment can consist of about 50k lines of very modular source code. (This estimate is based on a generous extrapolation from the 30k line size of the current UNICON system.) Clearly it would never be worth trying to factorise such a small core programme, with a view to multi-vendor implementation.

Finally it is noted in passing, for what it is worth, that the specifications for PCTE and CAIS interfaces are some 500 and 1100 pages respectively, per language binding.

- As outlined above, the UNICON conformance specification would probably run to about 10 pages - from which, together with some descriptive material, any group should be able to write their own conforming UNICON Environment, in the language of their choice.

7. DISCUSSION

7.1 *Does Unicon Include Sufficient Functions To Be An Adequate Core For A Software Engineering Environment ?*

UNICON is based upon a simple engineering notation, which is directly mapped onto a file-store to become a database with a graphics User interface. An additional reference system supports database segmentation and distribution, as an integral part of the version management function. However, even if the scope and integration of these important facilities are accepted, there remains some question as to whether they are in harmony with the provision of other necessary facilities.

- eg project management, security, E-Mail communications, User object-typing, application-specific rule-sets, etc etc.

It is taken as being self-evident that a Standard should confine itself to the bare minimum of concepts and detail, sufficient to meet its declared objectives - by focussing on essentials, it will broaden its scope.

- The UNICON conformance proposals are held to agree with this principle, with the sole declared objective being the transfer of application work-packages between UNICON environments, using the defined notation.

In following this minimal approach to Standardisation, the only rationale for extending the scope of the UNICON conformance definition, would be to achieve a desired degree of integration of some excluded facility, which could not readily be done by other means - eg using a handler.

Whilst it would be bold to say that UNICON as it stands already addresses all of the functions which have to be closely integrated, it is certainly the case that none of the areas already considered would come within this core-function category

- This situation is probably due to the extreme generality of the UNICON notation (with conventional ENTITIES/RELATIONS being available as a degenerate case of the extensions), the generic level of operation of the UNICON tools, and the flexibility of the ATTRIBUTE management arrangements.

- For example, it would be open to any UNICON implementation to define application-specific ATTRIBUTE fields, together with associated rule-sets - eg for ENTITY typing. This information would be compatible with the standard Import/Export format, where it would be transferred anonymously as ATTRIBUTE byte-stream data. (This would in no way be a shady practice, since communication between any two environments will always require some form of shared context - eg common knowledge of HOOD, ADA, etc).

In summary, it is argued that if UNICON were to become a Standard along the proposed lines, the areas not covered (such as Project Management etc) would become the subject of what is known in the US as an "after-market" of suppliers - without in any way compromising the Standardisation objectives.

- However there is probably a good case to be made for defining a small standard set of navigation-type database query functions, for use by loosely-coupled tools.

7.2 *"A Software Engineering Environment Is A Large Complex Piece Of Software Which Has Been Compared To An Operating System And A Database Rolled Into One"*

Whilst versions of the above statement find their way into the literature as throw-away truisms, the present paper has argued that such complexity need not be the case.

- There seems to be a paradox in which some problems can be almost intractable when considered in isolation - eg the design of a general Version Management function - yet when addressed within an appropriate context, a solution can readily be found which is both trivial to implement, and delivers an unexpectedly large functionality, due to mutual interaction within the context.

- The design of Software Environments seems to be particularly prone to the above paradox. The situation has been likened within the literature (Ref 4) to constructing a bridge across a chasm - the whole is very much greater than the sum of the parts. If you set out with concepts of insufficient scope, you will end up worse off than if you had not started in the first place.

There is no fundamental reason why a Software Environment having only 50k lines of code, cannot deliver a functionality well beyond any yet available.

7.3 *"The Introduction Of A Software Environment Into An Organisation Will Inevitably Require Significant Changes In Their Development And Management Procedures"*

It is now widely recognised that the introduction of new work-practices/tools to a large organisation, involves timescales measured in years rather than months. It may therefore be a pragmatic stratagem that UNICON's flexibility should be exploited in the first instance to emulate the existing working practices - warts and all.

- The notion is to regard the putting in place of the resources to support evolution, as a separate objective in its own right, to be done with the absolute minimum of disruption and change.

- After as long a period as it takes to achieve familiarisation, evolution could then begin - with an appropriate involvement and consensus of the whole community with regard to direction and pace.

- In the case of the UNICON reverse-engineering experiment reported above, it would be quite possible to introduce UNICON as say a simple Graphics tool and a Versioning aid to the existing development processes. In time, the automatic generation of the batch command files would be introduced - as a checking operation for the manual work .... and so on a step at a time.

As well as being able to precisely emulate existing methodologies and practices, UNICON also has significant rationalisation capabilities:-

- MASCOT is an elegant design/implementation methodology, of some twenty years standing, where asynchronous "ACTIVITY" threads communicate through "CHANNEL" letter-box buffers. From the UNICON perspective, it is argued that the designers of the later MASCOT3 sub-system extensions have been profligate in their coining of new language concepts and keywords, and have placed onerous demands on the User in requiring so many items to be given names. Since UNICON can provide a much more comprehensive set of sub-system constructors, in a general way which would none the less generate the desired MASCOT network (along the lines of Fig 7), it is held that many of the new MASCOT3 "concepts" can now be regarded as being to do with implementation rather than functionality.

- A not dissimilar point can be made with respect to the somewhat ill-defined HOOD methodology. The aim here was to introduce levels of hierarchy above the OOD code modules, through the introduction of pseudo objects - although they looked like normal objects from the outside, their services were actually provided by nested OOD objects. Once again UNICON can provide these hierarchical subsystem-type levels within a much more general context - which can of course include the more restricted HOOD psuedo-object formulation as a sub-set if desired.

8. FUTURE UNICON WORK

The aims of this paper have been to present a new approach to the through-life support of large software-intensive systems. This led to the development of an engineering description notation, and its subsequent application to Software Environment design, and to standardisation proposals.

The Defence Research Agency is continuing with the codification and evaluation of UNICON, for possible use by the Ministry of Defence. The importance of adhering where possible to widely accepted practices and standards is of course recognised - to this end the DRA invites comments and criticism of the UNICON work, and views on whether it is likely to attract a wider interest.

On the assumption that this work will eventually have a wider application, the Defence Research Agency would be interested to hear from any party which wishes to be involved in the evaluation, codification, or sponsorship of UNICON as a possible standard, or to have earlier access to the technology.

9. POSTSCRIPT

Dear Reader, if you believe that the development of the UNICON concepts actually followed the logical sequence outlined above, then to echo the reply by the Duke of Wellington when addressed by a stranger in his club as "Mr Smith I believe" - "if you believe that you will believe anything".

10. ACKNOWLEDGEMENTS

JOHN HARRISON is the chief programmer for the UNICON work. Progress to date is due in no small part to his expertise. Over the years valuable contributions, support, and criticism, have been made by many people including F DAWE, I DICKENSON, J PETHAN, A MULLEY, A PEATY, J EYRES, J BAKER, P KEILLER, V STENNING, M WEBB, S DOBBY. It has been a pleasure to work with such able colleagues.

11. BIBLIOGRAPHY

1 **Brown A W, Earl A N, McDermod J A, "Software Engineering Environments"** McGraw Hill Int Series in Software Eng, 1992

-- A timely review of Software Environments and Standardisation initiatives - this book provides a valuable background to the present paper.

2 **Booch G, "Software Components With Ada"** Benjamin/Cummings, 1987

-- An excellent ADA text which includes a good treatment of ADA limitations when applied to the top levels in large systems. Page 557 discusses the need for a sub-system abstraction to sit above ADA.

3 **Brown A W "Object-orientated Databases"** Mcgraw Hill Int Series in Software Eng 1991

-- An up-to-date review of the field. Of particular interest is the classification of Object Databases into different categories.

4 **Fisher A S, "Case - Using Software Development Tools"** Wiley & Sons, 1988

-- A good review of all the major CASE tools and methodologies. It notes on page 259 that most of the (then) current CASE methodologies "share the same underlying metaphor".

A GENERALIZATION OF THE SOFTWARE ENGINEERING MATURITY MODEL

Karl G. Brammer
ESG Elektroniksystem- und Logistik-GmbH
P. O. Box 80 05 69
D-8000 Muenchen 80
Germany

James H. Brill
Electro-Optical Systems
Hughes Aircraft Co.
P. O. Box 902
El Segundo, CA 90245
USA

SUMMARY

The software process consists of the methods, practices and tools used to generate a software product. The Software Engineering Institute at Carnegie Mellon University has developed a Capability Maturity Model (CMM) which defines five levels of maturity for the software process. Also included are sets of criteria that allow the specific assessment of actual software engineering maturity in given projects or organizations.

In areospace projects, software engineering very often is coupled with or embedded in systems engineering. It is therefore desirable to know if and how the CMM can be extended to systems engineering. The paper shows that this approach is feasible.

After a brief summary of the original Capability Maturity Model an overview and comparison of software and systems engineering disciplines is provided. Differences between software and systems engineering are highlighted and modifications are proposed to adapt and generalize the CMM accordingly.

Finally, the framework for a Systems Engineering Maturity Model is presented. This model is intended as a reference scale for systems engineering capability, in a similar way as the CMM applies to the software process.

1. GOALS

During the course of our activities in systems engineering we became aware of the work of the Software Engineering Institute (SEI) at Carnegie Mellon University. They have developed a framework for assessing the maturity of the software process in a given organization.

On the other hand, from our experience with many providers and users of systems engineering products and services in the public and private sectors of several countries, we knew that a corresponding method is lacking for the evaluation of systems engineering maturity. Thus we have aimed this article at those who are, as we, looking for a framework to evaluate and improve the competitiveness of the systems engineering process.

The goals of this paper are twofold. The first is to provide a framework for assessing systems engineering maturity and to identify the critical process areas in which improvement would raise the overall process to a higher level of excellence. The second goal is to stimulate further work in the development and application of methodologies to assess and improve the practice of systems engineering.

Before addressing these goals, a brief review of software and systems engineering is provided. This establishes a common basis for comparison of the two disciplines and serves as a bridge for the generalization of the SEI Capability Maturity Model from software engineering to systems engineering.

We are aware that there may be shortcomings in this attempt, but we are hopeful that our preliminary results will be accepted within the scope of this article.

2. REVIEW OF THE SEI CAPABILITY MATURITY MODEL

The software process consists of the methods, practices and tools applied in the course of a project to generate a software product. In November 1986, the Software Engineering Institute (SEI) at Carnegie Mellon University (CMU), with assistance from the MITRE Corporation, began developing a framework that would allow organizations to assess the maturity of their software process (Ref. 1). The initial framework soon evolved into a comprehensive maturity model (Ref. 2, 3).

Figure 1 shows a summary of the model. It is characterized by five levels of software engineering maturity, the lowest being level 1, Initial and the highest being 5, Optimizing. The underlying philosophy for the characteristics of the five levels is to look at the features of the software process in terms of:

- Process Procedures
- Process Performance
- Engineering Style.

The maturity of the process procedures is rated according to the degree of systematic approach, and of support by state-of-the-art tools and automation. At level 1 no formalized rules are practiced, at level 2 experience is passed on orally, at level 3 proven procedures are laid down in written form, at level 4 the emphasis is on process metrics and at level 5 a self-optimizing mechanism is reached for the process.

The maturity of process performance is judged by the deviation of the actual process results from the planned goals of productivity and quality. The higher the maturity level, the less is the risk, i. e. overruns become consistently smaller and estimates get more and more reliable.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

Maturity Level	Characteristics	Key Process Areas
5 Optimizing	Feedback: Process continuously improved	Defect prevention Technology innovation Process change management
4 Managed	Quantitative: Process measured *Focus on metrics*	Process measurement Process analysis Quality management
3 Defined	Qualitative: Process defined & institutionalized *Focus on process org.*	Organizational process definition Training program Peer reviews Intergroup coordination Software product engineering Integrated software management
2 Repeatable	Intuitive: Process dependent on individuals *Focus on proj. mgmt.*	Requirements management Software project planning and tracking Software subcontract management Software configuration management Software quality assurance
1 Initial	Ad hoc / chaotic: Process unpredictable	

Figure 1. Levels of Capability Maturity Model

The maturity of engineering style is often loosely characterized as progressing from (1) the "creative artist" via (2) the "tribal group", (3) a "corporate identity" and (4) "professional leadership" up to (5) the" software factory".

The Key Process Areas in figure 1 constitute the practices that must have been implemented in the software process in order to qualify for levels 2, 3, 4 and 5 respectively. There are no requirements for level 1.

The fully developed Capability Maturity Model (CMM) is augmented by a set of instruments for the actual assessment procedure:

- Description of Maturity Level Characteristics
- Description of Key Process Areas
- Maturity Questionnaires
- Respondent's Questionnaire
- Project Questionnaire.

At each level the appropriate key process areas are addressed by the Maturity Questionnaires. These questionnaires probe for the process characteristics of every level from 2 upwards. Usually a software producing unit begins its assessment with the test for level 2. Only when this test has been passed to complete satisfaction, the next test for level 3 will be conducted, and so on. No level may be skipped. The Respondent's Questionnaire is used to describe the professional background of the person completing the the maturity questionnaire. The Project Questionnaire is applied to characterize the project for which the maturity questionnaire is being completed.

The relationship of the various concepts used in this context is illustrated in figure 2 (Ref. 3).

The key process areas identify the requirements for each maturity level, i.e. they define the enabling practices. The process maturity reflects an organization's ability to consistently follow and improve its software engineering process, i.e. it indicates to which degree the enabling practices are actually implemented. The process capability is the range of results expected from following the implemented process, i.e. it predicts future project outcomes. The process performance characterizes the actual results achieved from following the process.

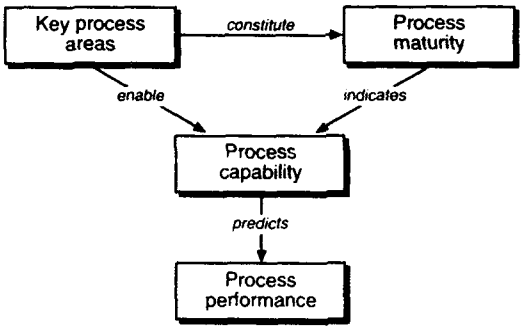


Figure 2. SEI Process Maturity Framework

3. SCOPE OF SOFTWARE ENGINEERING

Software consists of computer programs, procedures, associated documentation and data pertaining to the operation of a computer system. Software Engineering is the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of software (Ref. 4).

As we will see in section 4, software constitutes a subset of the set of basic elements of a general system. Almost absent in the systems of the early fifties, software quickly began to take over ever increasing parts of system functions. The growth of software was so fast that the "software crisis" developed (Ref. 5). As a consequence great efforts were triggered to drive software activities from a kind of creative art to a branch of engineering. In 1985 the DOD-STD-2167 on Software Development was issued, marking a similar milestone for software engineering as did the MIL-STD-499 for systems engineering.

A summary of the relevant concepts and terms of the current revision A of DOD-STD-2167 (Ref. 6) is given in figure 3.

In order to correlate with the scope of systems engineering, we refer to the statement in DOD-STD-2167A that it should be used in conjunction with MIL-STD-499 for total system development. In this context we have defined the scope of Software Engineering as shown in figure 4. (The figure as such is not part of 2167A, but has been set up to match figure 5).

The Software Elements under consideration are those mentioned at the head of this section plus the software elements of

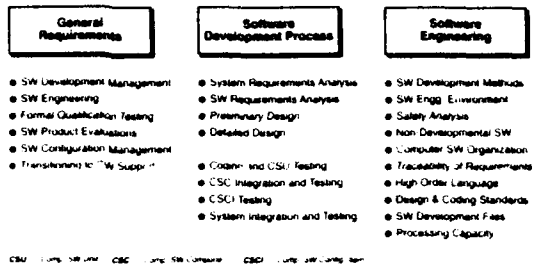


Figure 3. Summary of Software Development

firmware. The hardware elements of firmware are not included.

The input to all the tasks in figure 4 are the customer needs only in the case that the software system is a stand-alone deliverable item. Often the software is embedded in a wider system.

Then the input to the software tasks are the software requirements and applicable portions of the system requirements. These are the results of the first steps of the systems engineering process. The output of the activities in figure 4 matches figure 5, but is confined to software only.

Regarding the primary functions of figure 4, only Development and Support of software systems are explicitly mentioned in 2167A (besides acquisition). But taking into account the statement that it should be used in conjunction with MIL-

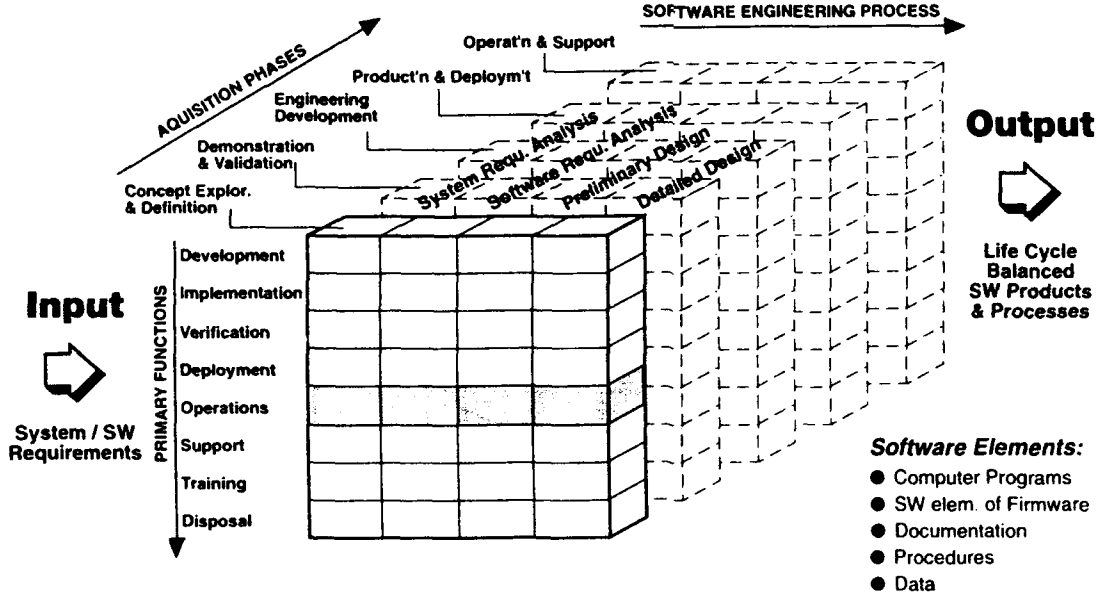


Figure 4. Scope of Software Engineering

STD-499, the primary functions of the latter standard have been carried over. Only the term "Manufacturing" of the overall system (meaning fabrication and assembly of test models as well as low-rate initial and full-rate series production) is replaced in this paper by the term "Implementation", i.e. coding and integration of the software system.

The term software engineering process does not appear in DOD-STD-2167A. This standard speaks of the Software Development Process and of Software Engineering, see figure 3. Looking at the eight steps of the software development process, one can associate the last four steps to implementation and test ("manufacturing" and "verification"). The first four steps bear a resemblance to the systems engineering process. They are therefore presented in figure 4 under the name of "Software Engineering Process".

The Acquisition Phases correspond directly to those of systems engineering. Only the development of manufacturing installations can be ommitted because the series production of software reduces to the simple act of copying.

4. SCOPE OF SYSTEMS ENGINEERING

The term of Systems Engineering originated only some decades ago in the aerospace field. During the sixties this discipline reached a high level of professional standard, promoted especially by such large-scale endeavours as the Apollo project. In 1969 the MIL-STD-499 on Engineering Management was developed by the USAF to form a first milestone in defining and unifying the associated basic concepts and procedures. The current revision A is directed at all of DoD systems engineering as well as joint Government-industry contracts (Ref. 7).

In these days, revision B of MIL-STD-499 is being prepared, with the modified title of Systems Engineering (Ref. 8). Accordingly, a System is an integrated composite of products, processes and people that provide a capability to satisfy a stated need or objective. Systems Engineering is an interdisciplinary approach to evolve and verify an integrated and life-cycle balanced set of system product and process solutions that satisfy customer needs.

MIL-STD-499B defines the scope of Systems Engineering as shown in figure 5. The physical scope of any system is delineated by the configuration of its System Elements, i.e. its basic constituents: Hardware, Software, Facilities, Personnel, Data, Material, Services, or Techniques.

For all of these system elements the Systems Engineer has to consider three aspects, forming the three axes of the cube in figure 5, i. e. Primary Functions, Acquisition Phases and the Systems Engineering Process.

The eight primary functions are those essential tasks, actions or activities that must be accomplished to ensure that the system will satisfy customer needs during the total life-cycle.

The five acquisition phases cover the evolution of the project, after the pre-concept tasks of assessing technological opportunities and formulating the customer needs, and before the post-operation tasks of decommissioning and disposal.

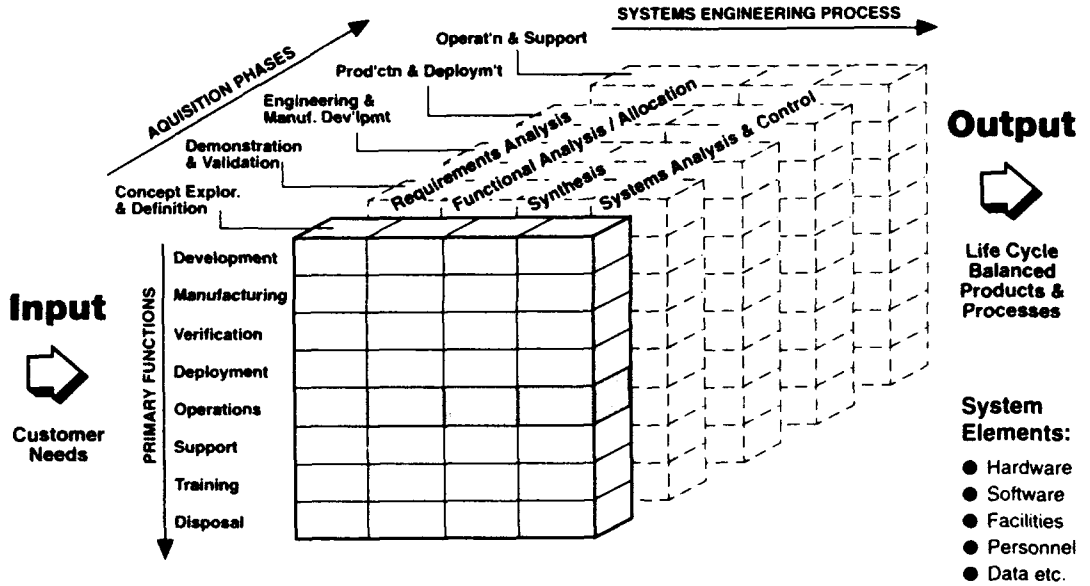


Figure 5. Scope of Systems Engineering

The four consecutive steps of the systems engineering process are Requirements Analysis (incl. definition of constraints, measures of effectiveness, functional and performance requirements), Functional Analysis and Allocation (incl. top-down decomposition of the functional architecture and requirements flow-down), Synthesis (i.e. the translation of functions and requirements into solutions in terms of the system elements), and Systems Analysis and Control (w.r.t. the measures of system and cost effectiveness and to the system configuration).

The systems engineering process is applied to all relevant primary functions of the system life cycle. The initial run of this process is started and performed in the first acquisition phase and is then iterated during each succeeding phase. Note that it is not sufficient to do the process only for the operational requirements and only once in the definition phase (as marked by the shaded bar in the foreground of figure 5), although this is usually considered to be the most important process part.

It is evident that the scope of systems engineering is quite large and that the required job is very demanding. Note that the above formulations follow the draft of revision B of MIL-STD-499. Compared to the current revision A the following extensions have been introduced:

- System Elements: "Procedural Data" have been replaced by "Data, Material, Services, or Techniques".

- Primary Functions: Development, Training and Diposal functions have been added. These will generate their own specific requirements, e.g. on CASE tools, training facilities, or materials that can be safely discarded or recycled.

- Systems Engineering Process: "Mission Requirements Analysis" has been generalized to "Requirements Analysis", "Functional Analysis" and "Allocation" have been combined into one step, and "Systems Analysis and Control" has been added. So, even if there still are four steps, the work involved has expanded considerably.

5. SOFTWARE ENGINEERING VERSUS SYSTEMS ENGINEERING

Software engineering, although having started later than systems engineering and suffering from a high pace of ever increasing volume, has made considerable progress since the software crisis was identified in 1968.

Meanwhile, software engineering has been catching up in formal procedures with systems engineering. In some aspects it even surpassed systems engineering. The authors became aware of this when they learned about the Capabilty Maturity Model (CMM). Such a model does not exist for systems engineering. We believe that it will be very beneficial to obtain a similar model for systems engineering, for several reasons:

- The CMM provides a standard by which each project, division or company can measure its professional competence in software engineering. From the detected deficiencies actions for improvement can be derived.

- Since the CMM already contains some key areas that are also part of systems engineering (requirements management, quality assurance, configuration management etc.) it may be considered as a basis for generalization.

- The scope of systems engineering is being more precisely defined and slightly extended by the new draft of MIL-STD-499B.

- The demands on systems engineering are becoming more challenging due to recent developments in the military and commercial acquisition environment (increased competition, lean production, design to life-cycle cost, concurrent engineering, CALS etc.)

- The resources of systems engineering are greatly improving due to more powerful and cost-effective development environments. Modern computer-aided systems/software engineering (CASE) tools allow to master more complex systems using elaborate formal system description and design tools.

- A framework for benchmarking and improving systems engineering will probably have a similar impact on raising the overall competence in systems engineering as the CMM does in the field of software engineering.

At the intent of generalizing the CMM from software to systems engineering, some caution is appropriate:

- Software engineering is only a part of systems engineering and this part is sometimes small, see figure 6.

- Software elements are usually only a subset of all system elements, see section 4.

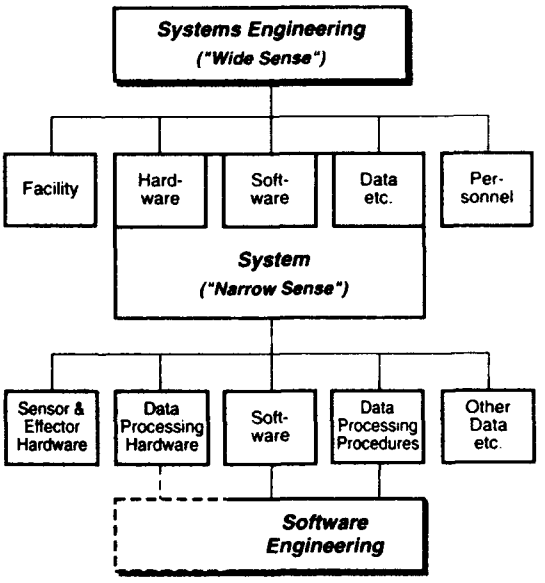


Figure 6. Systems vs. Software Engineering

- Hardware elements have features and processes different or absent from software elements. A good example is the problem of series production, i.e. copying the prototype system to full-rate quantities for deployment and fielding.

- The CMM is accompanied by detailed questionnaires to evaluate compliance with the capability requirements at each of the upper four levels of competence.

Nevertheless, the authors feel that the CMM is a good starting point for a Systems Engineering Maturity Model.

In the effort to transfer and generalize the CMM from software engineering to systems engineering we use the concepts of software engineering and systems engineering in the sense of DOD-STD-2167A and MIL-STD-499B (draft) respectively, see sections 3 and 4. We are aware that neither of these standards is universally recognized. But both standards are widely known, and they are practically applied or tailored for many projects. Thus they provide a valuable common language to discuss the systems and software engineering paradigms.

6. SUITABILITY OF THE CMM WITH RESPECT TO SYSTEMS ENGINEERING

The SEI Maturity Model has already been applied to many software organizations. So, on the one hand it has been realistically validated and, on the other hand it has efficiently contributed to the improvement of the software process in the assessed software producing units.

But no tool set is perfect and the CMM, too, has its detractors. Some critics have expressed concern that it fails to recognize the impact of competent and motivated individuals and multidiscipline teams on reducing risk and increasing productivity and quality. The SEI acknowledges that "the CMM is not a silver bullet and does not address all of the issues that are important for successful projects" (Ref. 9).

Nevertheless, the authors consider the software engineering CMM as a good basis for generalization in the direction of a corresponding model for systems engineering.

To test our premise that the CMM is indeed suitable for this purpose, SEI and nine of its Software Process Assessment Associates were requested to review and comment on our preliminary draft model.

Responses were received from SEI (informal), Pragma Systems Corporation, Dayton Aerospace Associates and American Management Systems. The three assessment associates provided encouragement and valuable written comments; all four respondents indicated that our premise was feasible. Their constructive contributions have been incorporated in the present article.

7. PROPOSED SYSTEMS ENGINEERING MATURITY MODEL

The inputs received from the three assessment associates and our own continuing work suggest that adaptations of the SEI model to systems engineering must go beyond those presented in this paper. However, we believe to have established a baseline from which one can proceed further to develop the Systems Engineering Maturity Model to the full extent.

The present status of the Systems Engineering Maturity Model (SEMM) is given by figure 7. The similarities, modifications and additions with respect to the Software Capability Maturity Model (CMM) are apparent from a comparison between the figures 1 and 7.

The basic features have been carried over unchanged from the CMM to the SEMM, because they are proven and apply equally to both software and systems engineering disciplines. Moreover, it was the intention of the authors to keep as much consistency as possible between both models. Therefore the principle of five maturity levels, the designation of the levels and the characteristics of the levels have been retained.

Changes have been entered to the key process areas by a transformation in three steps. First the KPA's have been generalized en bloc from software to systems.

Then it was checked for each KPA whether this step is valid with or without modifications. And finally, new key process areas have been added where deemed necessary. The additions are mainly due to the fact that systems engineering covers a wider scope than software engineering, see section 5.

The KPA's to be mastered in order to qualify for level 2 have been augmented by one area, i.e. System Risk Management.

For the level 3 KPA's the modifications are: extension of "Training" to include "Education", extension of "Reviews" to include "Testing", replacement of "Intergroup coordination" by "Interdisciplinary group coordination", generalization of "Software product engineering" to "Life-cycle balanced product engineering" (see figure 5) and of "Integrated software management" to "Integrated systems management".

Only minor modifications were made for the level 4 KPA's; the wording was chosen to be more concrete.

The first of the level 5 KPA's was generalized from "Defect prevention" to "System problem prevention", the second KPA carries directly over, and the third KPA was expressed in a slightly more general way. On level 5 the focus is on automation and integrated computer-aided systems engineering (not shown in figure 7).

Note finally that the benefit of "Increased Productivity and Quality" of the SEI model has been replaced by "Increased Customer and Producer Satisfaction" to reflect a more general view of quality and productivity. Systems engineering at a high maturity level delivers products and services which match the needs of the customer, and does so reliably within narrow goals of cost and time schedule.

Maturity Level	Characteristics	Key Process Areas
5 Optimizing	Feedback: Process continuously improved	System problem prevention Technology innovation Process management
4 Managed	Quantitative: Process measured *Focus on metrics*	Process mapping / variation Process improvement data base Quantitative quality plans
3 Defined	Qualitative: Process defined & institutionalized *Focus on process org.*	Organizational process definition Education and training Reviews and testing Interdisciplinary group coordination LC balanced product engineering Integrated systems management
2 Repeatable	Intuitive: Process dependent on individuals *Focus on proj. mgmt.*	System requirements management Project planning and tracking Subcontract management System configuration management Quality assurance System risk management
1 Initial	Ad hoc / chaotic: Process unpredictable	

Figure 7. Systems Engineering Maturity Levels

8. COMPLETING THE SYSTEMS ENGINEERING MATURITY MODEL

The full development of the Systems Engineering Maturity Model (SEMM) requires that the five instruments of the CMM, too, be generalized from software engineering to systems engineering.

Two examples for the description of the maturity levels 1 and 2 are shown in figures 8 and 9. These, too, are based on the SEI CMM and have been extrapolated for use with the new SEMM.

In the description of the key process areas it is necessary to incorporate systems engineering aspects that are absent from software engineering, e. g. the development of series production elements.

The adaptation of the SEI questionnaires on the respondent and the project should be straightforward.

However, the generalization of the Maturity Questionnaires will constitute the bulk of the remaining work. We understand that a major US company has developed a tailored version of one of the SEI questionnaires to assess its systems engineering integration and test process. The assessment team concluded that the adapted CMM and questionnaire was appropriate for this purpose (Ref. 10). Other organizations in the aerospace and defense sector are also proceding in this direction.

Level 1 – The Initial Process

- Unstable environment lacking sound management practices – Commitments not controlled.
- Success rides on individual talent and heroic effort.
- Standards and practices often sacrificed to management priorities – Usually schedule and / or cost.
- Process capability is unpredictable – Schedule, cost, and performance targets are rarely met.
- Level of risk is consistently underestimated.

Figure 8. Systems Engineering Maturity Level 1

Level 2 – The Repeatable Process

- Management discipline ensures that during schedule crunches systems engineering practices are still enforced.
- Basic system management discipline is installed.
- Previously successful processes are repeatable in a stable, managed environment.
- Achievable and measurable activitites are planned prior to making commitments – Commitments are tracked.
- Before schedule commitments are made, a process capability exists that enables the team to meet schedules.

Figure 9. Systems Engineering Maturity Level 2

Besides the material presented in sections 7 and 8 the authors have produced the preliminary characterizations for level 3 up to 5, but these are omitted in this paper.

Our own experience and initial effort, the encouragement received from many sides and the ongoing work in various organizations, including the National Council on Systems Engineering (NCOSE) indicate that a suitable version of a Systems Engineering Maturity Model can indeed be developed on the basis of the software-oriented Capability Maturity Model of SEI.

Following the presentation and publication of this paper we welcome further comments, especially from the software engineering community, to promote the complete development and finalization of the evolving SEMM.

9. CONCLUSIONS

In the initial section of this paper we stated two major goals:
1) to provide a framework for assessing systems engineering processes and to identify critical process areas
2) to stimulate further work in the development and application of methodologies to assess and improve the practice of systems engineering.

It is appropriate that our conclusions be related to these goals.

As to the first goal, we have approached it by developing an extension of the SEI software engineering framework. The resulting systems engineering framework reflects - besides the strenghts - also the weaknesses of the CMM, it is still incomplete, and we have treated the structural problems with insufficient depth. But even with these shortcomings we believe that the framework provides a first basis for systems engineering process assessment. Further, the key process areas discussed in section 7 and summarized in figure 7 address a number of critical problem fields of systems engineering. We are confident that improvements in these areas will indeed raise customer and producer satisfaction and also reduce the development risk of products and services.

As to the second of our goals, we have indicated in section 8 those parts of the framework that are still incomplete or missing, leaving opportunities for complementary work in this important area. During our research activities we became aware of a growing consensus among buyers and sellers of systems engineering products and services that sound methodologies to assess systems engineering maturity are actually needed and would be widely welcome. Given this need we would like to encourage interested collegues in joining the effort to finish a complete Systems Engineering Maturity Model.

In closing we wish to thank the AGARD Avionics Panel for the opportunity to present the results of our current work.

Acknowledgements

We acknowledge and give credit to the work of the SEI, to the contributors for the major revision B of MIL-STD-499 on Systems Engineering, and to the initiatives of both our companies to achieve increasing levels of software engineering and systems engineering maturity.

References

1. Humphrey, W.S. and Kitson, D.H., "Preliminary Report on conducting SEI-assisted Assessments on Software Engineering Capability", SEI/CMU Techn. Rep. SEI-87-TR, July 1987.
2. Humphrey, W.S., *"Managing the Software Process"*, Addison-Wesley, Reading, Massachusetts, 1989 (ISBN 0-201-18095-2).
3. Curtis, B., "Software Process Improvement Seminar for Senior Executives", SEI/CMU, sponsored by US DoD, 1992.
4. "IEEE Standard Glossary of Software Engineering Terminology", IEEE-STD-610.12, 10 Dec. 1990.
5. Naur, P. and Brian, R., editors, "Software Engineering", Report on a conference of the NATO Science Committee in Garmisch 1968, NATO Conference Proceedings, 1969.
6. "Defense System Software Development", DOD-STD-2167A, 29 Feb. 1988.
7. "Engineering Management", MIL-STD-499A, 1974.
8. "Systems Engineering", MIL-STD-499B, Draft 1992.
9. SEI/CMU Techn. Rep. SEI-91-TR-24, 1991.
10. "Systems Engineering Integration and Test Capability", internal questionaire, draft version 1.0, LORAL, 10 Aug. 1992.

Author's Biographies

Karl G. Brammer is presently head of the central engineering process group at Elektroniksystem- und Logistik-GmbH in Munich, Germany. His previous assignments at ESG include systems engineering for a wide range of military and civil systems. Prior to joining ESG in 1972, he held R&D positions at Dornier and the German Aerospace Research Organization. Dr. Brammer holds an MS degree in Aeronautics and Astronautics from MIT and a Doctor's degree in Electrical Engineering from the Technical Institute of Darmstadt. He is a member of the US National Council On Systems Engineering (NCOSE) and the German societies DGLR, DGON and VDI.

James H. Brill is an internationally recognized expert and lecturer in the fields of systems engineering and program management. He is presently a Division Manager, Hughes Aircraft Company, Electro-Optical Systems. His previous assignments within Hughes include Program Manager, M1A1 and Manager, Systems Engineering Laboratory. Mr. Brill has lectured and presented seminars on systems engineering at Hughes and throughout the United States, Canada, England, Australia and Europe. He is the author of the Manager's Handbook of Systems Engineering, and holds degrees from Syracuse University, New Mexico State University, and the National War College. He is presently a member of the board of directors of NCOSE.

Discussion

Question W. ROYCE

Will the definition of a systems-oriented capability maturity model force a change in the current SEI model for software only?

Reply

This question raises a good point. Looking beyond our present work, I would not exclude the possibility that the authors of the CMM will review their model with respect to systems engineering. One area that comes to my mind would be to incorporate in the CMM the interfaces of the software engineering process to the systems engineering process.

APPLICATION OF INFORMATION MANAGEMENT METHODOLOGIES FOR PROJECT QUALITY IMPROVEMENT IN A CHANGING ENVIRONMENT

Fabrizio SANDRELLI
(SW Quality Responsible)
ALENIA DAAS
Via dei Castelli Romani,2
00040 Pomezia (Rome) ITALY

Summary

In the technologically advanced field of avionics, quality is today required both by the market and the engineers. This means that quality is not only a goal to reach new costumers, but is a different way of working which aims to reach a higher and more satisfactory working environment.

The attention has been focused on the technical information produced during the development of the project and its diffusion through the technical and managerial environment. The goal is the improving of the quality of the whole development process where the quality of the final products comes as a consequence.

This paper describes a methodology experienced in the last few years in ALENIA (Pomezia Plant) to plan, achieve and manage a more flexible and advanced way of working, through the strong involvement of all who contribute to the realization of the product.

Attention has been focused both on the organizational aspect of the project and the product configuration. Those two aspects are self related and bring to a correct management of the project.

List of Symbols

CI	Configuration Item
CPJ	Company Project
CSCI	Computer Software Configuration Item
HWCI	Hardware Configuration Item

Information Structuring

The capability to maintain software projects and to guarantee their quality, is based on the way the internal communication of the technical and technical-managerial information is managed. All the information about process and product has been structured and the communication process organized and automated.

Task of every project is to produce the technical documentation that will allow the building of the product (be the product a software code as well as a physical device). So, from the project development point of view, the technical documentation is the product. A correct quality and configuration control of the technical documentation allows then to know exactly the status, the story and the evolution of the project, in order to achieve a much better quality of it.

A complex Project, during its life-cycle phases, is hierarchically decomposed, following a top-down methodology, into elementary projects, each of ones is finalized to the development of one or more configuration items.

At high level the design process is independent from the technology and is represented by a generic Configuration Item (CI). Only at the lower levels it differentiates in hardware and software components, each of ones is typically finalised to the development of one or more CIs. The CIs whose functionalities have been exactly defined are always referred to as HWCI if they will be developed as hardware and CSCI if they will be developed as software.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

In this paper, mostly interested to software, a CSCI is considered as an aggregation of software that: has been requested to be a CSCI by the purchaser, needs a separate development, my be considered by itself, has a high level of complexity and a high probability of evolution, is general-purpose and may be used in more applications, satisfies an end use function, is designated for separate configuration management and has a direct impact on the budget.

The software project/product life-cycle is managed at CSCI level: the technical documentation is produced and the planned quality activities (reviews, acceptance tests and audits) are performed for each CSCI.

Through a set of project/product relationships a hierarchical structure of activities is obtained (Work Breakdown Structure), whose terminal elements are the configuration items related to the project.

In fig.1 there is a typical hierarchical decomposition of a complex project (CPJ) into elementary projects (PJs); each CPJ structure starts horizontally from its founder and lasts with the development of the related items.

In this structure, every CPJ is organised in "Project Activities" (PJA) following the "activity based management" techniques. Each PJA is composed of PJs, each of ones is finalised to the development of one or more Configuration Items. Every project is planned to be articulated into phases with a standard production of technical and managerial documentation and information about responsibilities, goals and progresses of times, costs and results.

The structure which binds up the PJs and the relative managerial documentation to the configuration items is called "Project Tree".

Task of the CPJ leader is to guarantee the technical and economical control of the CPJ itself. This is obtained assigning one responsible and a budget to every PJA that belong to the CPJ. Every PJA reports the activity work packages, including costs (hours, technical equipment, general expenses, suppliers and so on) and development times.

At the end of the estimating phase, such activities are structured in a hierarchical tree and constitute the Work Breakdown Structure of a complex project. Both goals and the tasks achieved are associated to each PJA. The responsible for the activity is then able to know the matching or the deviations against the goals.

At the end of the development of a CPJ, the set of PJs activated for any specific development, has structured in a hierarchical relationship all the CIs, CSCIs and HWCIs that compose the whole product,

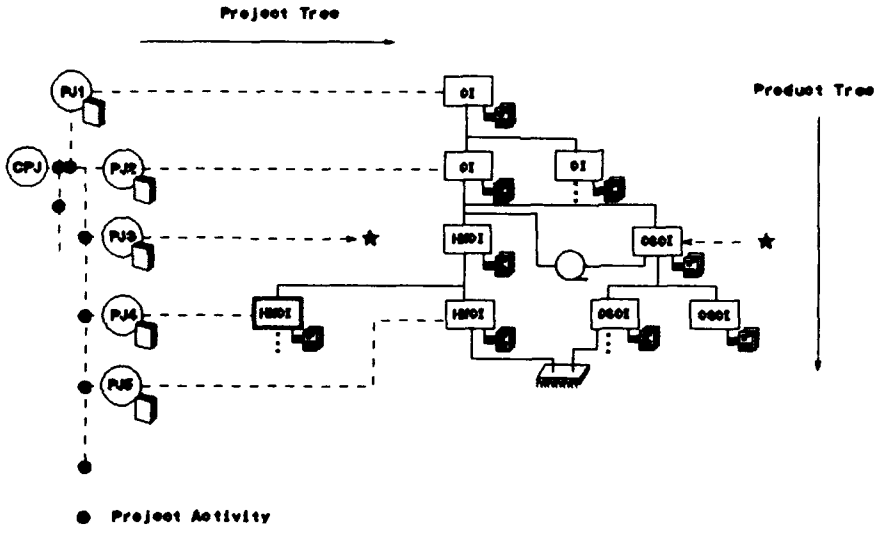


fig.1 - Configuration Tree

and the technical documentation has been developed to the most detailed components.

The structure composed of the whole of the CIs, CSCIs and HWCIs and the technical documentation is called "Product Tree".

The set of the software configuration items is structured in a sub-set of the Product Tree called "Software Tree".

At the beginning of a software project, the software project leader and the software configuration controller create in the Central Data Base a Software Tree for it. In the Software Tree, a software project/product is assigned a CSCI identified by a code. This CSCI is decomposed with the top-down methodology, in lower level CSCIs with a higher degree of detail, which are also identified by an univocal code.

If, for example, an operative system is a "special purpose" one, it is assigned a low level CSCI which hierarchically belongs to the CSCI of the software project into which the operative system is developed.

On the other hand, if the operative system is a "general purpose" one, it is assigned one dedicated CSCI, but it can be used in any software project. This can be done because the software may be imported from one CSCI to any other one. In the case of the example, the software files that compose the general purpose operative system may be imported, for being used, in any project that needs them.

Any low level CSCI is articulated into non-configured components (CSC), each of ones is also assigned a code. Each CSC is an aggregation of Computer Software Units (CSU), that is: files, as shown in fig.2.

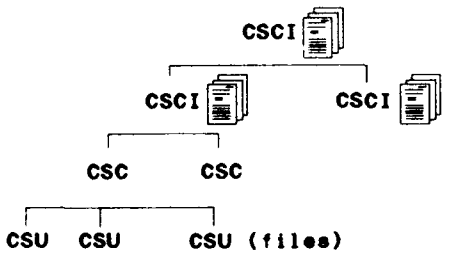


fig.2 - Software Breakdown Structure

Associated to each CSCI is the technical documentation which constitutes the outputs of the software product life cycle. When such documents are actually placed under formal configuration control, they have already been given all the authorisations and successfully passed all the reviews.

This way of structuring allows working in parallel on the same software project. If, for example, the product will be a computer board with a microprocessor on it, the software project will be composed of the basic software for driving the microprocessor and the application software which drives the application. Assigning one CSCI to the basic software and another CSCI to the application software, allows two different software teams to start working in parallel, having defined only the interfaces between the two CSCIs. When the basic software will be requested, for example for loading it into the microprocessor, the software stored under the basic software CSCI will be imported into the CSCI which needs it at the requested release, also if the basic software has evolved in the meanwhile to subsequent releases.

This concept of linking different CSCIs or, at higher level, different CIs, is essential for managing large scale projects which involve subcontractors, as it usually happens. It is of fundamental importance to manage the subcontractors technical documentation in the same way as the internal one, in order to have a real estimate of the whole project progressing. For this reason, all the documentation form the subcontractors is also placed under configuration control.

The technical documentation is the link between the PJA and the software product (as shown in fig.3). At any time of the software project, looking at the archived documentation, it is possible to know what has been produced by the Company and by the subcontractors, which means that it is possible to "measure" the progress status of the project. There are two indicators that allow a correct management of the activity: the quantity of documentation produced against the estimated and the degree of confidence of this documentation. The confidence is guaranteed by the quality activity, because every document may be stored into the Data Base only if

authorised by the quality responsible after the appropriate reviews. The quality head also monitors the progress of the project in order to point out every delay in the documentation production and take the appropriate actions.

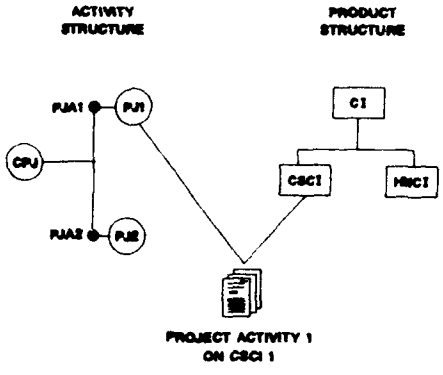


fig.3 - Activity/Product Relationship

Any problem in the developed software or in the documentation is managed through a Problem Reporting and Corrective Action system. If there is a need for a change, a software item under configuration control may be updated after careful evaluation and by authority received from the provided heads, following the change flow described in fig.4.

The instigators of software changes may be internal or external to the Company; they detail the reasons for change requests, using standard or non standard forms.

Automation

It has been accepted that any formal procedure helps in achieving the goals. The project structures are physically implemented into an automated Information System which gives the measure of the progress of the project through the actual status of the documentation stored.

The Information System is based on a Central Data Base, into which it stores the information that come from the progress of the project activities and from which it takes all the data about planning and costs. From a conceptual point of view, all the information are spread when formally stored into the Central Data Base, considering that the Central Data Base can be accessed through a network of local workstations by any area of the factory.

The use of the Information System involves the use of informal procedures that have been standardised and automated. This is a basic requirement for producing data in an orderly and controlled way and to have them available in time. These procedures allow to aggregate all the data about responsibilities,

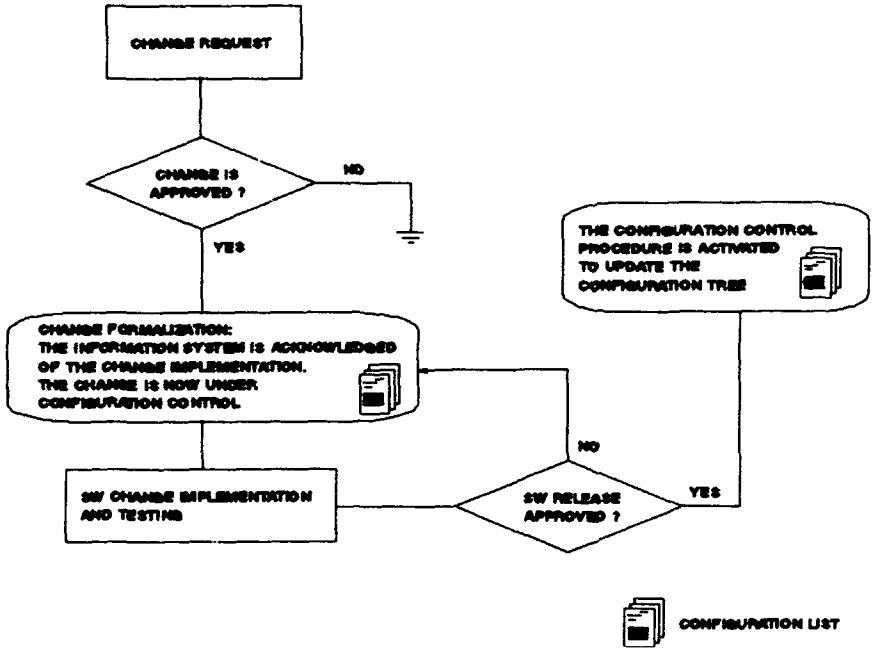


fig.4 - Changes Management

planned documentation and scheduling to the different levels into which a complex project is articulated.

The use of the Information System allows to:

- rationalize all the technical documentation in all the phases of the project life-cycle,trace the anomalous situations and manage the product configuration;
- enhance the exchange of technical and managerial information between different areas and the availability of information since the beginning of the project;
- have a constant and capillary tracking on the project activities, in order to activate a real time corrective action procedure, if necessary.

The Information System maintains and guarantees the configuration integrity (organization, control, status accounting) relatively to the technical product documentation of the configuration hardware and software items (HWCI and CSCI) and to the technical data of interest in the development, like information structures, data base architecture etc.. It also allows to:

- manage accurately and in a standard way the hardware and software product and project documentation (both electronic and on paper);
- parallelize the project activities in a controlled way;
- manage hardware and software interactions.

The Information System provides the tools necessary to a good management of the product development.

During the development of the project, the Information System gives the global situation of the progress of the project itself through a the set of reports, for example the "Current " and "Historical" Status Reports, the "Where Used" and the "Review and Audits" ones.

The use of these reports allows to anticipate as much as possible the spread of the information necessary to the development of the whole project, in such way to manage all the related Activities as much efficient as possible.

Data from the Information System are presented in tabular forms, the most used of which is the Configuration List. An example is in fig.5 where a software tree is reported.

Through the Configuration List all the changes to the configured items are managed.

When a change is needed and agreed in the software already developed, the CSCI is given the status of

LEV.	CODE	REL.	REV.	DESCRIPTION	RESPONSIBLE
1	M067RA	2.0		HL CSCI	
-	M0674RA0-01AFT	2.0		Absolute File on Tape	
-	M0674RA0-01SDP		A	SW Development Plan	
-	M0674RA0-01SQP		A	SW Quality Plan	
2	M0674RA1	1.1		LL CSCI	
-	M0674RA1-01SRS		B	SW Requirement Spec.	
3	M0674RAA	1.0		CSCI A	
-	M0674RAA-01SRS		A	SW Requirement Spec.	
-	M0674RAA-01SDD		B	SW Detailed Design	
4	M0674RAA011		1	CSC 011	
5	M0674RAA011S001		2	CSU S001	
3	M0674RAB	1.1		CSC B	
-	M0674RAB-01FFH	1.1		Fusing File	
4	M0674RAB05A		4	CSC 05A	

fig.5 - Configuration List

"changing in progress". This status results in the Configuration List, because the release number of the affected CSCI is printed between "()".Who makes a printout of the report knows that the software is going to be changed, the foreseen date when the changes will be applied and who is the responsible for this action.

As a demonstration, it may be considered that often a software change has an immediate impact on the Production Department. The Production Head knows, from the Configuration List, that the software actually in use is going to be changed, he must immediately consider whether continuing to use the old software or suspending the production, waiting for the next software release. From the Configuration List he takes all the information needed to support his decision.

Such a way of proceeding favours the diffusion of the information horizontally through the company, reducing to the minimum level the necessity of organizing work progress meetings and reducing a lot the time between the moment when the critical event occurs and when the new situation is faced.

Conclusions

This way of working allows to create information in a standard and accurate way and to speed up as much as possible the information diffusion and updating; it favours the people who are involved in the project to take always more part in it, promotes team-works, enhancing the responsibilities capillarily down to the lowest levels of the developing chain.

The timeliness by which information is available, allows the parallel start of developing phases otherwise sequential, with a cascade process which anticipates a lot the end of a project, compared with the traditional developments.

Such a dynamic and effective way of managing the project, makes anomalies and errors more evident so that they can be solved as early as possible. The final product is intrinsically improved from the quality point of view,because the intervention is spread along the whole developing process with the important contribution of all the involved departments and, at the same time, costs and developing times are dramatically reduced.

The Discipline of Defining The Software Product

John K. Bergey
Software and Computer Technology Division
Systems and Software Technology Department
Naval Air Warfare Center, P.O. Box 5152
Warminster, Pennsylvania, 18974-0591 U.S.A.

1. SUMMARY

Under the sponsorship of the Naval Air Warfare Center Warminster, Pa., U.S.A. an initiative was undertaken to establish a *de facto* project standard for a "software product" for Mission Critical Computer Resources (MCCR). This effort is an outgrowth of a risk reduction approach to improving the MCCR software development and acquisition process based on W. Edward Deming's renowned quality principles.

While defining a "software product" might appear to be rudimentary, the hypothesis is put forth that the software product itself, which has been inexplicably neglected by the software engineering community, is the key element and focal point for understanding and improving the software development and acquisition process.

This paper describes the model of a "software product" that was developed to 1) promote uniform software product nomenclature, 2) serve as a reference point for software process improvement, 3) provide a basis for developing a more cost-effective software documentation standard, and 4) provide a more rigorous means of specifying software deliverables for MCCR. Other potential benefits of the software product model include providing a convenient and effective means to 1) uniformly baseline diverse software products, 2) establish software configuration management and control, 3) ensure the capture of all software components required for regeneration, 4) provide a natural mechanism for the collection of software product metrics, and 5) facilitate the on-line sharing and reuse of software programs, documents, and data. The software product model that has been developed is referred to as SPORE, which stands for Software Product ORganization and Enumeration. The SPORE is a graphical, hierarchical model that provides a complete and logical decomposition and description of a software product for MCCR. The productized version of SPORE will be suitable for the uniform specification, procurement, and configuration management of software products for software life cycle support. The ultimate goal is the evolutionary development of a full-blown SPORE that will be a complete Software Product Open Repository/Environment.

2. THE SOFTWARE EXPLOSION

The defense and aerospace industry is experiencing a software explosion as a result of the widespread availability of low-cost, high performance computers. This software explosion has manifested itself in various forms: 1) a diversity of computer system architectures, operating systems, programming languages, software engineering environments, and communication networks, 2) a proliferation of Computer Aided Software Engineering (CASE) tools and software application domains, and 3) a wide spectrum of development methodologies and software innovations, such as 'object-oriented design' and 'graphical user interfaces', to meet the need for more capable and user-friendly software. However, despite all the new technology developments in hardware and software, the demand for software is rapidly outstripping our ability to produce it. And the elusive goals of producing error-free software, on schedule, and within budget, remain elusive and formidable.

3. MEETING THE SOFTWARE CHALLENGE

What will it take to meet the software challenge? First, it will take an abundance of profound knowledge. The starting point is recognizing that ...

Software Development is a highly creative, arduous and abstract, labor-intensive process of great complexity. The process itself is characterized by successive and highly iterative stages - each of which constitutes a radical transformation - that together produce an invisible end-product. The outputs of the individual stages are one or more descriptive documents and/or intermediate software elements (either of which may, or may not, signify completion). The stages typically progress from a set of specified needs and constraints to a computer system architecture to a set of software requirements to a software design to software code units to a set of executable elements that implement a particular instantiation of the desired functional capability which (hopefully) meets the specified needs within the given constraints.

Second, it will take perseverance and commitment to bring about the necessary transformations in our current policies, practices, and engineering culture that traditionally have been hardware-oriented.

The comprehensive definition of software development that is given above serves to provide needed insight into the scope and complexity of the software development process that we must master, or that will continue to master us. Clearly, there is no 'silver bullet' solution to the software problem on the horizon. And perhaps, as the

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

definition above implies, a uniform, formal discipline solution may not be achievable in the classical engineering sense. There is, however, the promise of steady, if unspectacular progress [1].

4. IMPROVING THE CURRENT STATE-OF-PRACTICE IN SOFTWARE

With the astronomical growth in software, it is widely recognized that there is an ever increasing need for uniform, disciplined approaches to advance the cause of software from an art form to an engineering science. This applies not only to the software development process, but to the entire software life cycle, which encompasses everything from major software enhancements and upgrades to corrective, perfective, and adaptive software maintenance. Consistent with W. Edward Deming's renown quality principles [2], an effective strategy for realizing steady software progress is to concentrate on the singular goal of improving quality (in areas of greatest identified risk) using a disciplined, process-oriented approach. This approach is believed to offer the most potential for coping with the growing diversity and complexity of software *and* for enhancing our ability to produce a quality software product in a timely, efficient, and cost-effective manner. The Capability Maturity Model (CMM) for software process improvement, which was developed by the Software Engineering Institute (SEI), is a well recognized example of such a disciplined, process-oriented approach to software risk reduction and quality improvement [3].

5. THE MISSING ELEMENT: "THE SOFTWARE PRODUCT"

Considerable emphasis has been placed on software process improvement as an effective means of improving software quality and productivity. Clearly, this has recognized merit. However, if software process improvements are to be properly directed and focused, a parallel effort is needed to explicitly define what constitutes a properly constructed and quality "software product" (i.e., the output of the process). While defining a "software product" might appear to be rudimentary, the argument is put forth that the software product, which has been inexplicably neglected by the software engineering community, is the key element, and, in fact, the focal point for understanding and improving the software engineering process. With the growing complexity of software, the diversity of application domains, and the inherently abstruse nature of software, there is a pressing need to establish a common software structure and unambiguous nomenclature. This is considered essential for identifying, describing, and communicating the salient characteristics and attributes of all the components that makeup a complete "software product". One value in creating an engineering model of the "software product" is its ability to serve as a uniform means of gaining visibility and insight into the complexities and interdependencies of the software development process. And how can one embark on improving and optimizing the software acquisition process (or the software development process) without first defining its output (i.e., the software product)?

6. THE SOFTWARE PRODUCT - A COMMON FOCAL POINT

The "software product" provides a *common focal point* across the acquisition process. Figure 1 depicts the basic software acquisition process that typifies how software is procured for Mission Critical Computer Resources (MCCR). The process comprises three major stages or phases: 1) the specification and procurement phase, 2) the contractual software development phase, and 3) the life cycle support phase. As indicated on the diagram, a key element in the process is the deliverable software product. A comprehensive understanding of the software product can serve as a focal point for improving the software development process, planning for software life cycle support, conveniently specifying software deliverables, defining software documentation needs, establishing sound software configuration management practices, or collecting software product metrics. A "software product" model is multi-faceted and can be viewed from many different perspectives as illustrated by the following descriptions:

1) a common, intuitive, software template,

2) a generic specification for the software product deliverables,

3) a hierarchical decomposition of the software and its documentation,

4) a standard engineering reference model with uniform nomenclature,

5) an interface standard for software deliverables,

6) a generic framework suitable for defining a repository for MCCR software.

7) a means to facilitate sharing and reuse of programs, documents, and data,

8) a rigorous model for the development of a software documentation standard, and

9) a training aid for educating management/ project personnel.

In actuality, the concept of a generic model for a "software product" embraces all of these ideas. As the list above suggests, there appear to be many practical applications for a "software product" model.

7. A CUSTOMER CENTERED, RISK REDUCTION AND QUALITY IMPROVEMENT INITIATIVE

This paper describes an initiative to develop a comprehensive model of a "software product" intended for MCCR. The initiative embraces a disciplined, customer-centered, process-oriented approach toward improving the current state-of-practice in software engineering. The potential benefits and the engineering rigor a model can provide were a significant impetus for undertaking the development of the "software product" model.

For the software customer who buys a Commercial Off-The-Shelf (COTS) software tool, the software product can be characterized as a floppy disk (that contains the

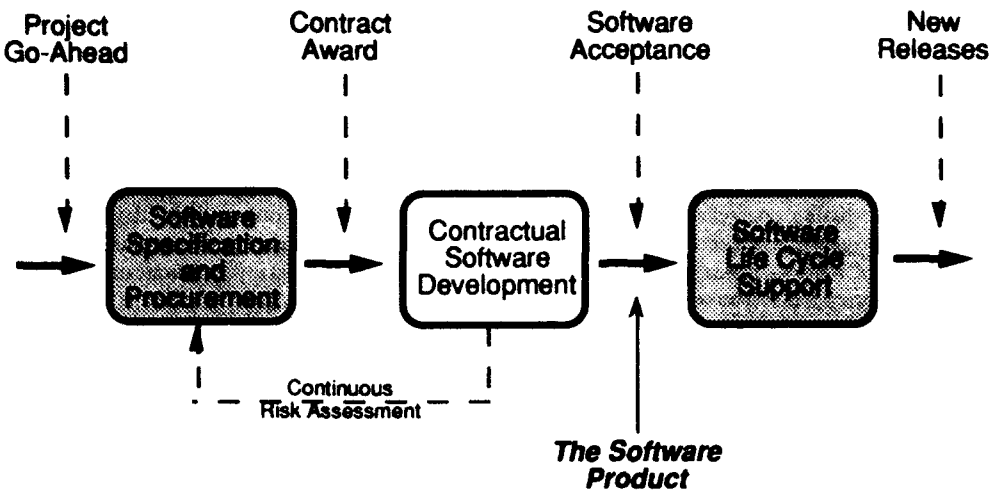


Figure 1. A Strategic Framework: The Software Acquisition Process

software application) and a set of user manuals or reference documents. The customer, in this case, does not have a need for anything more (in the way of a software product), and depends on the original vendor to support the product and periodically provide new releases to correct software "bugs", improve performance, and provide new and enhanced capabilities. The customer's concerns are mainly directed towards buying the most capability for the dollar that will meet his/her particular needs in their domain of interest.

In marked contrast, the customer who acquires software for MCCR requires a very different kind of software product because of the need to provide indigenous life cycle support for as long as the software product is operationally deployed. In this case, the software customer wants assurance that his/her activity (or company) will be able to take delivery of the software product and ensure a smooth transition to the life cycle support phase. This translates into a stringent set of requirements (in the procurement phase) to ensure obtaining a complete set of software deliverables, up-to-date descriptive software documents, information for regenerating the software end-product, and a clear understanding of the software product's structure, composition, dependencies, and interrelationships. The high skill level and discipline required to properly specify these requirements and understand all of the intricacies of a software product for MCCR were a major impetus behind this initiative to develop a comprehensive model of a software product.

The software product initiative addresses an area of significant risk that relates directly to a demonstrable customer need. As such, it represents a quality improvement that promises a significant return on investment. This initiative is believed to have significant potential for improving the current state-of-practice in software because the software product definition will provide valuable insight for the implementation of improvements to the processes associated with each of the three phases of the acquisition process.

8. THE SOFTWARE PRODUCT - MORE THAN A SET OF DOCUMENTS

A common practice in evaluating or monitoring the software development process, or judging the efficacy of a software product, or conducting a critical software review is to treat the software product as if it were synonymous with the associated set of descriptive software documents. This approach is flawed because there is no intrinsic mechanism for insuring that the descriptive software documents and the various elements that comprise the actual "as-built" software are truly reflective of each other. Software, therefore, is more than a set of descriptive documents or specifications - it is the sum total of all the elements that constitute a software product. These elements include the actual application programs and run-time software, program libraries, operating system software, software regeneration elements, software development artifacts, as well as the end-product and its comprehensive set of descriptive software documents. In essence, the software product encompasses all the intermediate products and by-products of each of the iterative stages described in the definition of software development.

9. THE QUALITY PAYOFF: HUGE COST SAVINGS

The potential cost savings that can be achieved by adopting a common model to understand, describe, specify, and procure software for MCCR are believed to be enormous. Today's major software standards do not adequately address the full extent of required deliverables. For example, they do not explicitly specify the software elements needed for software regeneration or the software development artifacts needed for efficient and cost-effective life cycle support. Consequently, every software project must fend for itself to obtain the extraordinary

expertise that is required to adequately specify the software product and its associated deliverables. The problem is that most projects do not have sufficient time or the expertise to do this properly. In fact, the commonly accepted practice of tailoring software standards and their deliverables for project-specific use often compounds the problem. Consequently, the delivered software product is rarely complete. Missing documents, missing source code, missing system generation directives, and an inability to regenerate the software are typical problems that may take months to years to rectify. The bottom line is that the typical project does not get delivery of the software elements and technical information necessary to load, install, operate, regenerate, maintain, and re-engineer the software. Even small projects (i.e., on the order of 20,000 Delivered Source Instructions) can expend 1-2 man-years of effort in preparing for life cycle support of the software, and large software projects (on the order of 100,000 Delivered Source Instructions) can expend many times this amount of effort. And in many instances, additional funds are required to procure software tools, software licenses, or even computer systems due to unforeseen events stemming from inadequate specifications and other technical considerations relevant to life cycle support. Occasionally, the problems escalate and litigation is necessary to resolve discrepancies and misunderstandings or differences of interpretation concerning the contractual software deliverables. Multiply this avoidable loss of time, man-power, and money across every software-intensive project and the amount becomes staggering. It is the summation of all these incurred costs across a very large number of projects that is projected to be enormous - in excess of hundreds of millions of dollars or more. Work is required to precisely quantify and document the quality-induced cost savings for a pilot project(s). However, based on years of experience working on software-intensive projects, the potential cost savings and return on investment are considered to be very substantial and warrant refining the model and possibly pursuing its adoption as a standard for MCCR.

10. REVERSE ENGINEERING: A SYMPTOM OF THE PROBLEM

In the current software literature there is a great deal of emphasis on re-engineering and reverse engineering. Re-engineering can be described as the middle ground between *repair* and *replacement* where the re-engineering is intended to revitalize an existing software system by making substantial changes in terms of adding new capabilities, adapting to new requirements, and/or improving supportability. It is understood that a well documented baseline is a prerequisite for re-engineering an existing system. However, the majority of operational systems in the field are typically undocumented, any specifications that do exist are out of date, and in many instances the program structure is virtually unknown, the code is unreadable, and the actual algorithms and software design characteristics are poorly understood, if at all. Consequently, before any re-engineering can be accomplished, it is usually necessary to "reverse engineer" a system to document its capabilities, structure, design characteristics, algorithms, and operation. While reverse engineering efforts can and do succeed, they typically require an enormous amount of time and effort that could otherwise be avoided if the systems were properly structured and judiciously documented in the first place. Reverse engineering is symptomatic of a problem - not a solution to it. In other words, the real answer isn't developing bigger and better ways (and tools) to reverse engineer systems, but to concentrate on ways of acquiring (with the original system) an effective and affordable set of baseline documents and software components that are comprehendible and maintainable for the life of the software product. The amount of money that is being spent on reverse engineering of software products is indicative of the huge savings that could be realized through the application of a common "software product" reference model.

11. THE SOFTWARE PRODUCT FOR MISSION CRITICAL COMPUTER RESOURCES

This paper describes a generic model of a "software product" for Mission Critical Computer Resources (MCCR). The application to MCCR is emphasized to contrast it with software products intended for Management Information Systems (MIS), Automated Information Systems (AIS), and/or customary business or scientific applications. One major focal point for developing next-generation real-time systems is the operating system [4]. The distinguishing characteristics of a software product designed for MCCR revolve around the special operating system and run-time software features needed to support the time-critical processing requirements of real-time computer systems. It is the author's contention that the definition of the software product intended for MCCR is more stringent than that for non real-time software applications (e.g., MIS and AIS). Consequently, the approach that has been taken is to develop the software product model around MCCR applications, which are considered to have more stringent requirements. An appropriate subset of this model could be adopted by the MIS and AIS communities.

12. THE SOFTWARE PRODUCT MODEL: SPORE

The software product model that is described in this paper is referred to as the SPORE, which stands for *Software Product ORganization and Enumeration*. As depicted in Figure 2, the Spore Model was implemented on a Apple® Macintosh® personal computer using TopDown®, which is a COTS tool for automating system and process design and documentation. TopDown® was chosen because it is a very intuitive tool that facilitates the breakdown of a complex problem into small manageable parts, where each of the components can be expanded to show greater detail. The following sections of this paper are devoted to describing the goals, organization, composition, and characteristics of the SPORE Model.

®Apple and Macintosh are registered trademarks of Apple Computer, Inc. ®TopDown is a registered trademark of Kaetron Software Corp.

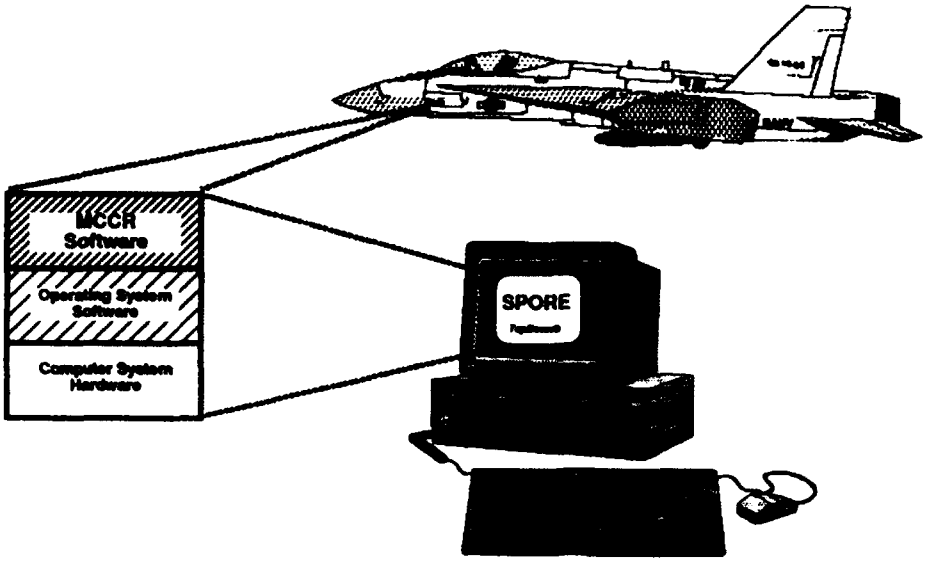


Figure 2. The Software Product for Mission Critical Computer Resources

13. THE SOFTWARE PRODUCT TOPOLOGY

The SPORE Model covers the entire software product topology and not just the software product itself. The software product topology is the term given to describe the complete software product and its operational environment and support environment. As shown in Figure 3, the Software Product Topology consists of the MCCR computer system configuration, the software product itself, and the software tool resources (tools and manuals) required for its life cycle support. By definition, it includes all of the information, tool resources, and/or software elements needed to 1) install and operate the software product in its operational environment, 2) regenerate the software end-product, and 3) perform life cycle support in an efficient and cost effective manner.

14. SPORE: SOFTWARE PRODUCT ORGANIZATION AND ENUMERATION

The SPORE can succinctly be described as a graphical, hierarchical model of the software product (topology) for MCCR. It provides a logical decomposition and description of all the elements that constitute the software product along with the other elements that are essential to its deployment and life cycle support. This initial SPORE model is considered a suitable engineering reference model for the uniform specification, procurement, configuration management, and life cycle support of software products for MCCR. However, as discussed in a latter section, this initial version of the SPORE needs to be thoroughly field tested using actual software product data obtained from projects.

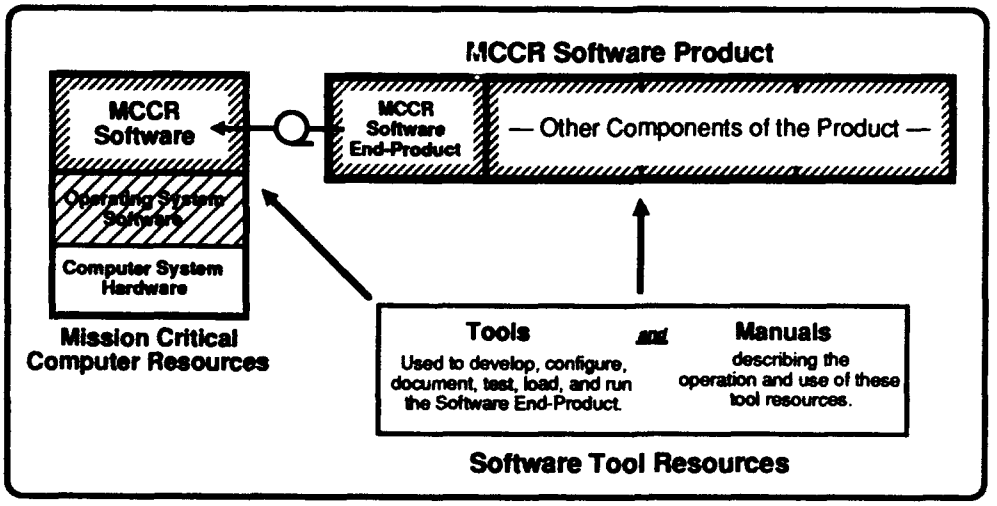


Figure 3. Software Product Topology

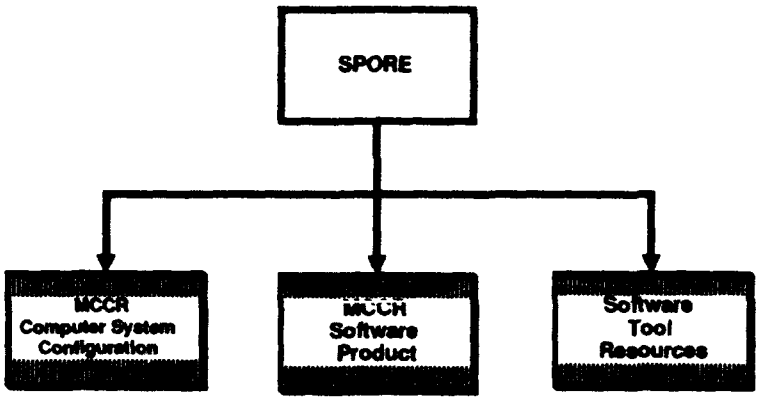


Figure 4. Top-Level Elements of the SPORE Model

The SPORE Model is not intended to constrain a developer or a project in any way. Rather, the intent is to provide a superstructure, or reference framework, for organizing and enumerating the elements that comprise the software product. Hopefully, any project can use the SPORE, or a subset thereof, as a common reference model (i.e., a software template) to identify and describe the configuration and contents of their specific software product. One of the purposes in developing SPORE was to establish uniform nomenclature and a common means of viewing a software product to gain visibility into its specific characteristics. The specific goals for the development of the model were the following: 1) *intuitive nomenclature*, 2) *logical breakdown*, 3) *understandability at top levels by management/project personnel*, 4) *understandability at lower levels by software engineers*, and 5) *complete specification of required components*. By definition, the required components were purposely limited to only those absolutely necessary for regenerating the software end-product, performing life cycle support in an efficient and cost effective manner, and installing and operating the software product in its operational environment.

15. SPORE MODEL DESCRIPTION

As shown in Figure 4, the top-level elements of the SPORE Model are 1) the ***MCCR Computer System Configuration***, 2) the ***MCCR Software Product***, and 3) the ***Software Tool Resources***. Together, these three top-level elements describe the entire software product topology, while the MCCR Software Product element describes the software product itself. These three top-level elements and their sub-elements are described in the following sections under the appropriate heading.

15.1 MCCR Computer System Configuration

The MCCR Computer System Configuration element defines the specific hardware and software resources of the operationally deployed MCCR system for which the software product was designed to operate. As shown in Figure 5, the three sub-elements of the MCCR Computer System Configuration element are 1) the *COMPUTER SYSTEM HARDWARE*, 2) the *OPERATING SYSTEM SOFTWARE*, and 3) the *OTHER APPLICATION SOFTWARE*.

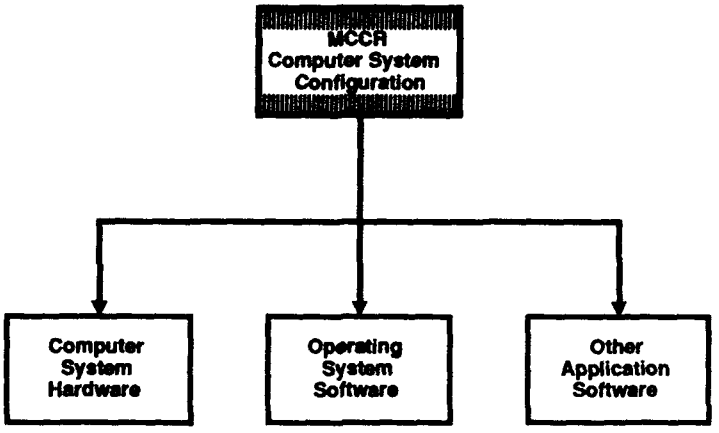


Figure 5. The MCCR Computer System Configuration

15.1.1 Computer System Hardware
The Computer System Hardware defines the particular hardware features of the deployed computer system that are required for the proper operation of the software product. This sub-element typically includes such items as the specific computer make and model, the memory size, auxiliary hardware items (e.g., floating point or memory management unit), and peripheral equipments such as hard disk drives, storage devices, communication equipment, and other special gear and/or computer system interfaces the software product must be compatible with or is dependent on.

15.1.2 Operating System Software
The Operating System Software defines the specific version of the operating system software, if any, that is installed on the system. Typically, this is the operating system software that is provided by the computer hardware manufacturer. However, the standard operating system software that usually comes with the computer system may not necessarily be used in MCCR applications. For instance, in MCCR systems using the Ada Programming Language, the run-time software is frequently a customized software system consisting of a separably procured Ada run-time kernel with an extensive set of software extensions that are developed to provide the necessary time-critical, real-time software capabilities.

15.1.3 Other Application Software
The Other Application Software defines application programs, or utilities, or other software that the software product being described must co-exist, interface with, and/or be compatible. However, the software described under this sub-element is not a part of the software product, although it may possibly be separably described as another software product that shares the same hardware resources.

15.2 The MCCR Software Product
The MCCR Software Product element provides a uniform and generic structure for identifying and specifying the particular components and attributes of a software product designed for MCCR. As shown in Figure 6, the four components of the MCCR Software Product are 1) the *MCCR SOFTWARE END-PRODUCT*, 2) the *SOFTWARE REGENERATION COMPONENTS*, 3) the *SOFTWARE DEVELOPMENT ARTIFACTS*, and 4) the *SOFTWARE PRODUCT DOCUMENTATION*. Together, they include all the software elements, data, and documents necessary to 1) regenerate the software end-product, 2) perform life cycle support in an efficient and cost-effective manner, and 3) install and operate the software product in its operational environment.

15.2.1 MCCR Software End-Product
The MCCR Software End-Product is the nomenclature used to refer to the physical software entity (typically encoded on a tape, cartridge, or other hardware media) that is ultimately loaded and installed on the MCCR computer system. This is the executable or operational form of the software product that provides the particular functionality and capabilities for which the software product was designed and developed. The end-product constitutes a particular release or instantiation of the software product, but it cannot be directly modified or maintained, because it is coded in a rigid form dictated by the computer system's instruction set architecture. As shown in Figure 7, the MCCR Software End-Product is comprised of *EXECUTABLE CODE ELEMENTS*, *EXECUTION SUPPORT ELEMENTS*, and *LOAD-AND-EXECUTE INSTRUCTIONS*. The *EXECUTION SUPPORT ELEMENTS* are further decomposed into *Library Elements*, *External Data Elements*, and *Operating*

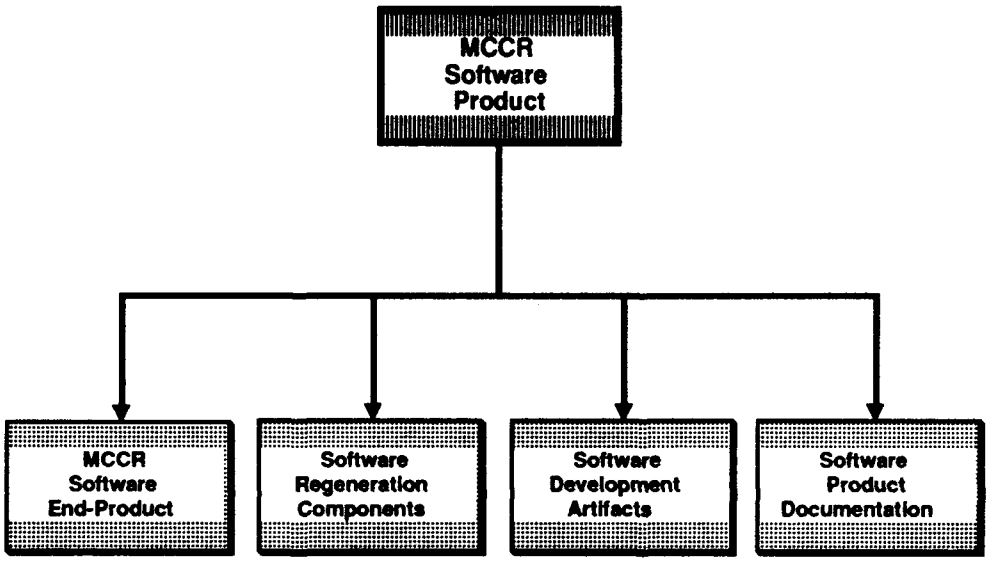


Figure 6. The Four Components of the MCCR Software Product

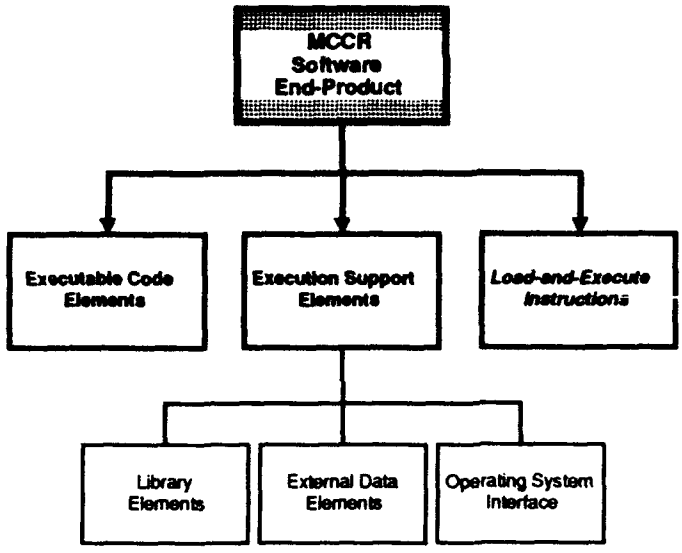


Figure 7. The MCCR Software End-Product

System Elements (which identifies the dependencies, if any, on the MCCR Operating System Software). The *LOAD-AND-EXECUTE INSTRUCTIONS* describe how the MCCR Software End-Product is to be loaded and installed into the computer system. Occasionally, the software end-product may be burned into Read Only Memory (ROM) or some other form of permanent or semi-permanent storage, in which case the software is referred to as "firmware". In this case, the *LOAD-AND-EXECUTE INSTRUCTIONS* would describe the procedure for physically installing the firmware device, while the actual "burn-in" of the firmware device would be described in a special section of the Software Regeneration Guide that is required as part of the software product documentation.

15.2.2 Software Regeneration Components

The Software Regeneration Components consist of those components and only those components that are necessary to recreate the end-product starting from the source code. The regeneration components produced by each stage of the software regeneration process are included to enable step-by-step verification and validation of the entire process. The regeneration process, itself, is described in the Software Regeneration Guide that is required as part of the software product documentation. As shown in Figure 8, the generic software regeneration components include *SOURCE CODE Regeneration Elements*, *OBJECT CODE Regeneration Elements*, *INTERMEDIATE LIBRARY Regeneration Elements*,

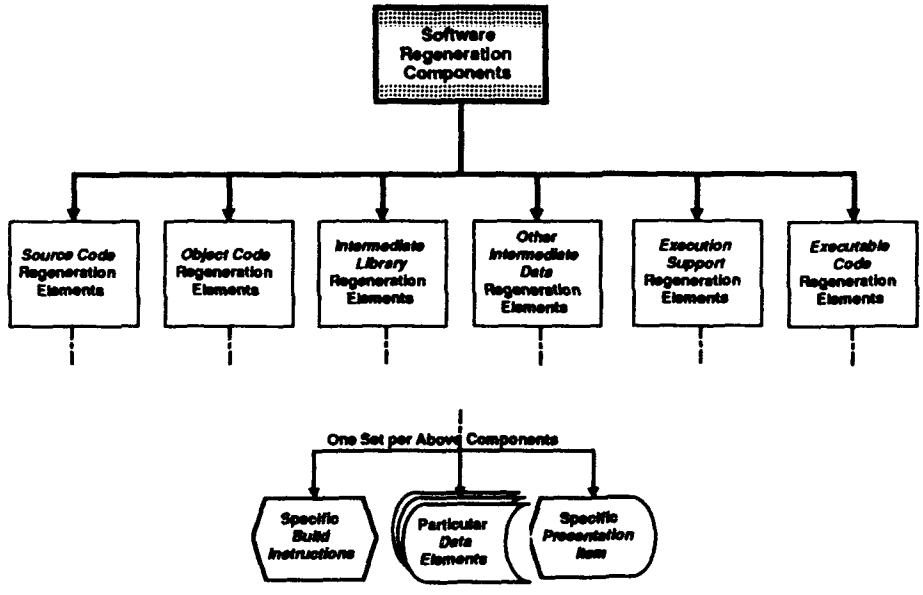


Figure 8. Software Regeneration Components

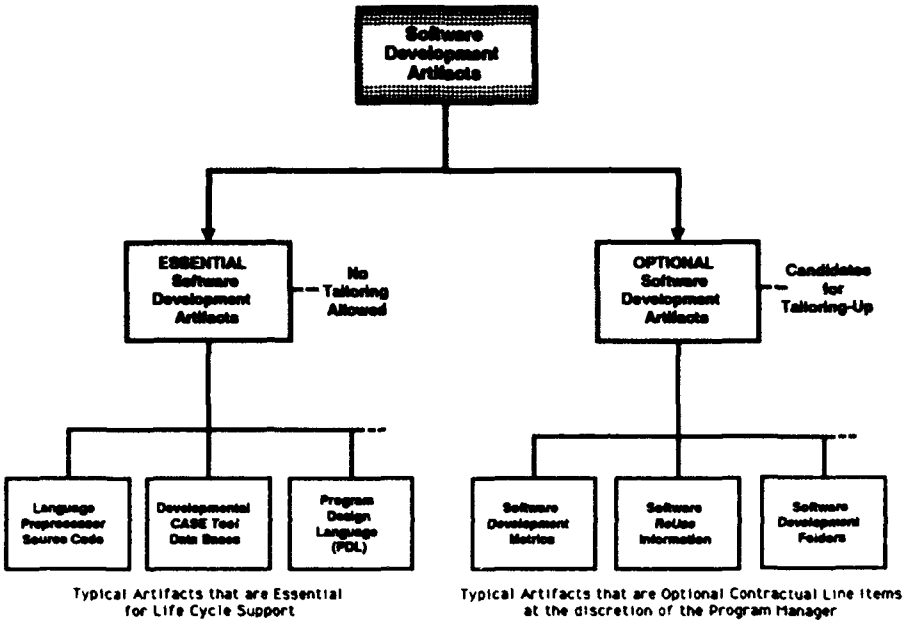


Figure 9. Software Development Artifacts

OTHER INTERMEDIATE DATA Regeneration Elements, *EXECUTION SUPPORT Regeneration Elements* (Libraries and External Data), and *EXECUTABLE CODE Regeneration Elements*. Each of these individual software regeneration components is broken down into three sub-components: 1) the specific *Build Instructions* that describe the procedure for creating the regeneration component, 2) the specific *Data Elements* that makeup the regeneration component, and 3) the specific *Presentation Information* for displaying and/or printing the particular regeneration component.

15.2.3 Software Development Artifacts

The Software Development Artifacts are software development entities or by-products that were produced and used in the original software development process but are not required for regeneration of the software end-product. By definition, these artifacts are divided into two groups: 1) *ESSENTIAL SOFTWARE DEVELOPMENT ARTIFACTS* and 2) *OPTIONAL SOFTWARE DEVELOPMENT ARTIFACTS*, as illustrated in Figure 9. Although the artifacts may have played a significant role during the development phase they may, or may not, be of particular value in the life cycle support phase. Subsequently, those artifacts that are considered to be essential to performing software life cycle support in an efficient and cost-effective manner are designated as being 'ESSENTIAL'; all others are designated as being 'OPTIONAL'. While the exact determination is somewhat subjective, some artifacts are clearly essential and some are obviously optional. Two examples of such artifacts are Program Design Language (PDL) descriptions and Software Development Metrics, respectively. If a PDL description of the software design was produced during the development phase it would definitely be a very valuable artifact to those responsible for maintaining and enhancing the software (and thus designated 'ESSENTIAL'). Although software metrics produced during the development phase may be valuable in providing status information they would hardly be essential to performing software life cycle support (and thus designated 'OPTIONAL') since they represent past history. It should be noted that just because an artifact is designated 'OPTIONAL' does not mean that it isn't valuable or that it shouldn't be a contractual requirement. The SPORE approach was predicated on defining the minimum required to perform software life cycle support in an efficient and cost-effective manner so that life cycle support could not be compromised through tailoring, contract negotiations, or any other means. However, the approach does allow and, in fact, encourages a project to "tailor up" their software requirements to meet specific needs by specifying any number of 'OPTIONAL' artifacts. This has the added benefit of being able to separately cost them out as optional line items in a contract so that their cost effectiveness can be determined on a case-by-case basis. For example, a set of regression tests for the software product could be specified as an optional artifact to be separately costed out. Based on the proposals and bids that were submitted, the procuring activity could determine if the development contractor should produce the regression tests, or whether it would be more cost-effective to hire an Independent Verification and Validation (IV&V) contractor to develop them, or have their own project personnel develop the regression test suite.

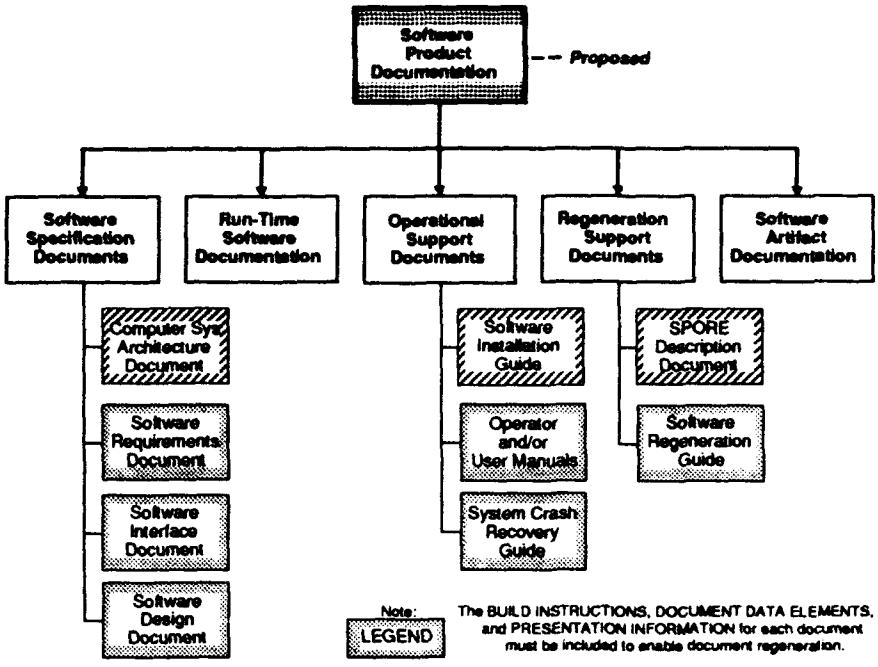


Figure 10. Software Product Documentation

15.2.4 Software Product Documentation

The Software Product Documentation consists of a suite of software documents that describe the software system architecture and requirements, its capabilities and operation, how the software is designed and constructed including its interfaces, the steps required to regenerate the software end-product, how to install and operate the software product in its operational environment, and other pertinent information on performing life cycle support. Three ground rules that apply to all of the documents are that they must: 1) conform to the minimum information requirements specified for each document (in any appropriate format that aids understanding and readability), 2) be delivered in an electronically readable and processable form and include a reproducible hard-copy document, and 3) include a copy of the *Build Instructions*, *Document Data Elements* (Text and Graphics), and *Presentation Information* that are necessary to regenerate each of the documents and create a hard-copy. The Software Product Documentation suite constitutes a "straw man" set of documents for a software product. The specific documents being proposed were derived from knowledge gained from performing an in-depth critique of the deficiencies of existing software documentation standards and from developing the SPORE Model of the software product. As shown in Figure 10, the Software Product Documentation is divided into five categories: 1) *SOFTWARE SPECIFICATION DOCUMENTS*, 2) *RUN-TIME SOFTWARE DOCUMENTATION*, 3) *OPERATIONAL SUPPORT DOCUMENTS*, 4) *REGENERATION SUPPORT DOCUMENTS*, and 5) *SOFTWARE ARTIFACT DOCUMENTS*.

15.2.4.1 Software Specification Documents

The Software Specification Documents are analogous to the classical software descriptive documents and include a *Computer System Architecture Document*, a *Software Requirements Document*, a *Software Interface Document*, and a *Software Design Document*.

15.2.4.2 Run-Time Software Documentation

The *Run-Time Software Documentation* describes the specialized MCCR system software which provides all, or part, of the real-time Application Program Interface (API) with the computer hardware resources. The run-time software typically includes an executive, system utilities, interrupt handlers, I/O device drivers, etc. A separate document was considered essential for the run-time software because it has the most stringent design considerations, the highest degree of complexity, and represents the greatest maintenance challenge.

15.2.4.3 Operational Support Documents

The Operational Support Documents include a *Software Installation Guide*, a *Operator and/or User Manual(s)*, and a *System Crash Recovery Guide*. These documents describe the man-machine interface and the basic system software features that are required to support a system operator and/or end-user in the operational environment.

15.2.4.4 Regeneration Support Documents

The Regeneration Support Documents include a *SPORE Description Guide*, and a *Software Regeneration Guide*. These documents identify the particular SPORE components for the software product

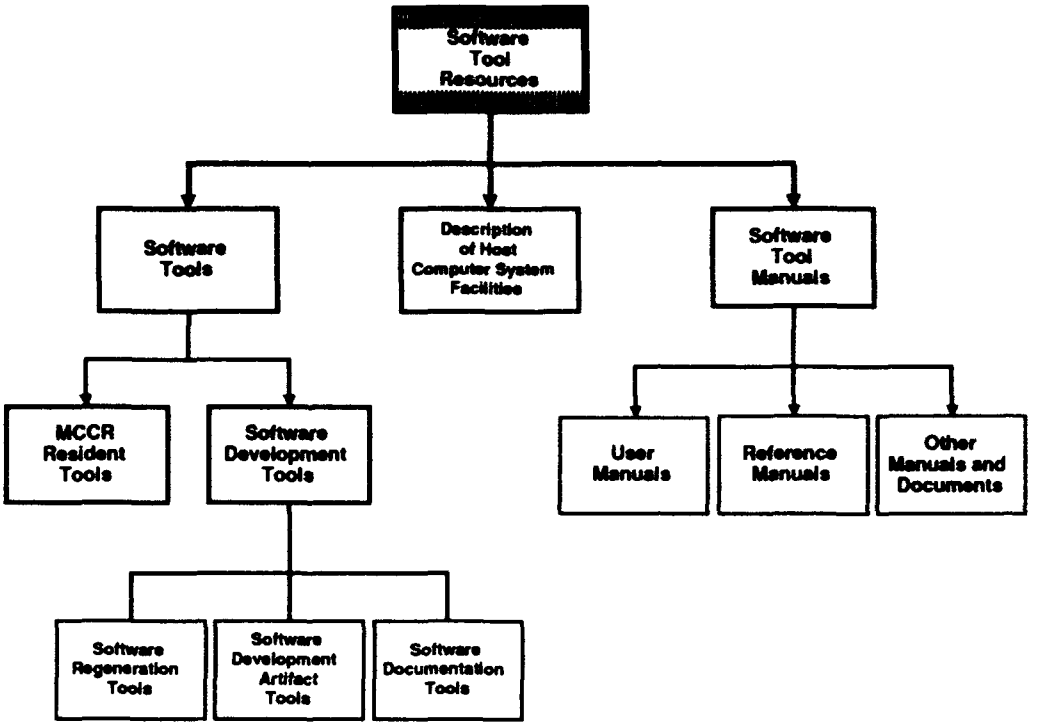


Figure 11. Software Tool Resources

deliverable and the process for regenerating the software end-product form the data and information elements contained in the SPORE, respectively.

15.2.4.5 Software Artifact Documentation

The Software Artifact Documentation includes documentation on each of the *ESSENTIAL Developmental Artifacts* and *OPTIONAL Developmental Artifacts* describing the specific nature and characteristics of the artifact, how they are used, the associated tool(s), relevant examples, and pertinent information on the artifact development process.

15.3 Software Tool Resources

The Software Tool Resources element identifies all the resources (i.e., software tools, facilities, and manuals) that were used to develop the software product and which are required to provide life cycle support. As shown in Figure 11, the three sub-elements of the Software Tool Resources are 1) the *SOFTWARE TOOLS*, 2) the *DESCRIPTION OF HOST COMPUTER SYSTEM FACILITIES* and 3) the *SOFTWARE TOOL MANUALS*. Together, they provide the information on the tools, descriptive tool manuals, and host computer systems (on which the tools reside) that is required to 1) regenerate the software end-product, 2) perform life cycle support in an efficient and cost-effective manner, and 3) install and operate the software product in its operational environment.

15.3.1 Software Tools

The Software Tools sub-element includes all the tools that were used in the development of the software product *and* are required to provide life cycle support. This includes the tools that were used to produce the software documentation. As shown in Figure 11, the *SOFTWARE TOOLS* sub-element is comprised of *MCCR Resident Tools* and *Software Development Tools*. The *MCCR Resident Tools* include any tools that run on the actual MCCR computer system and are used to support the loading, installation, or operation of the software end-product. The *Software Development Tools* are further broken down into *Software Regeneration Tools*, *Software Development Artifact Tools*, and *Software Documentation Tools*, corresponding to three (of four) components of the MCCR Software Product. It should be noted that if any of the required tools is also a developmental item, as opposed to a non-developmental COTS tool, it must be delivered as a separate software product (in conformance with the SPORE Model), itself, so that the procuring activity will have the means to perform life cycle support should that become necessary.

15.3.2 Description of Host Computer System Facilities

The purpose of this sub-element is to provide an overview and a summary of the computer system facilities that were used to host the tool resources used in the development effort *and* which are still

applicable for supporting the prescribed regeneration tools, artifact tools, and documentation tools specified above.

15.3.3 Software Tool Manuals

The Software Tool Manuals sub-element includes all the off-the-shelf documents that are available from the tool manufacturer to describe its use, operation, capabilities, and application. As shown in Figure 11, the Software Tool Manuals sub-element consists of *USER MANUALS*, *REFERENCE MANUALS*, and *OTHER MANUALS AND DOCUMENTS*. These three generic categories are provided to accommodate the high degree of variability in tool manuals and tool documents.

16. FUTURE SPORE DEVELOPMENTS

Future SPORE developments are broken down into near-term and long-term development efforts as described below.

16.1 Near-Term Development Effort

The current SPORE Model is a developmental version. It needs to be "productized" before it can be considered ready for use by projects. Consequently, the next step in the development of SPORE will be to thoroughly field test the model and refine it using "real-world" software product data. Several candidate, software-intensive projects of significant size and complexity have been identified that can serve as suitable testbeds. Another significant task that remains to be completed is the development of the detailed specifications for the proposed software product documentation suite.

16.2 Long-Term Development Effort

The ultimate goal that is envisioned is the evolutionary development of a full-blown SPORE that would be a complete Software Product Open Repository/ Environment that is built around an open-systems architecture. Such a system could provide a common means of gracefully delivering and transitioning software products from one government activity to another, from a sub-contractor to a system prime contractor, or from a contractor to a government activity. In terms of capabilities, this ultimate SPORE system would support the following scenario: 1) a complete software product is delivered on a single, high capacity, optical disk, 2) the optical disk is inserted into the system and the software product is loaded into SPORE, 3) a software manager accesses SPORE from a workstation and obtains on-line access to the delivered software product, 4) a modern Graphical User Interface (GUI) enables the manager to view a graphical decomposition of the software product on a display monitor, 5) the manager invokes a command to obtain a standard set of software product metrics, 6) the SPORE system responds by displaying a summary report of the software product which includes statistics on the software end-product, the number of individual program modules (including a listing of their name, source code count, programming language used, etc.), and the number of descriptive software documents (including a listing of their title and page count), 7) the software manager initiates a Configuration Management (CM) assessment capability to check the status of the software product, 8) the SPORE's CM system displays a list of items that are missing in the software product and flags numerous inconsistencies and several potential problem areas, 9) the manager invokes a Software Quality Assurance (SQA) analysis capability, 10) the SPORE's SQA system performs complexity measures on all the software modules and reports back on the modules that have exceeded a pre-specified limit, 11) the manager makes an inquiry about a specific program module, 12) the system responds by displaying the complexity graph for the selected module, 13) the manager requests a summary report, 14) the system produces a hard-copy summary of all the inspections and tests that were performed during the session, 15) the manager creates a backup and provides the authorization to allow the SQA Team (which is responsible for acceptance) to access the software product, and 16) the manager suddenly wakes up and remembers there is "another meeting" to go to, and that this dreaming - *about how things could be* - has to end!

17. ACKNOWLEDGMENTS

The Author wishes to acknowledge the contributions of both Dr. Donald J. Bagert of Texas Tech University, Lubbock, Texas who worked on the SPORE Project during the Summer of 1992 under the U.S. Navy-ASEE (American Society for Engineering Education) Program and Mr. Frank Prindle of the Naval Air Warfare Center (NAWC), Aircraft Division, Warminster, Pa. without whose help and technical expertise this effort would not have succeeded. The author also wishes to thank the many individuals in the Software and Computer Technology Division of NAWC Warminster who contributed to the group discussions and critiques.

18. REFERENCES

[1] Brooks, F. P. "No Silver Bullet", *Computer*, Vol 20, No. 4, April 1987, pp. 10-19.

[2] Deming, W. Edwards, "Out of the Crisis", August 1992, Massachusetts Institute of Technology, Center for Advanced Engineering Study, Cambridge, Mass. 02139 (ISBN 0-911379-01-0) pp. 18-24.

[3] Paulk, M.C., Curtis, B. , Chrissis, M.B., et al., "Capability Maturity Model for Software," Software Engineering Institute, CMU/SEI-91-TR-24, August, 1991.

4] Stankovic, John A. "Misconceptions About Real-Time Computing". *Computer Vol. 21, No. 10* (October 1988), pp. 10-19.

Discussion

Question - J. BART

Have you applied SPORE to a program? Are there future programs on which the SPORE approach will be applied?

Reply

The concept of SPORE has been applied to projects indirectly in the form of a software data base that was an integral part of a tightly coupled Software Engineering Environment (SEE).

The current SPORE effort is a new development and the SPORE model needs to be "productized" using actual project data before it can be considered really for general project application. Two candidate software-intensive projects of significant complexity have been identified that will serve as suitable test beds for this purpose.

The long range goal is to investigate the feasibility of using SPORE as the basic building block for a Software Product Open-architecture Repository/Environment (SPORE).

SDE's FOR THE YEAR 2000 AND BEYOND: AN EF PERSPECTIVE.

D.J.Goodwin
British Aerospace Defence (Military Aircraft Division)
Warton Aerodrome
Warton
Preston PR4 1AX, UK

1 SUMMARY.

The process of selecting a Software Development Environment for the embedded software of a large, complex military aircraft project can be long and costly.

This paper describes the process adopted on the European Fighter Aircraft project (EFA) by British Aerospace (BAe) from the initial research and prototyping exercises performed in the seventies through to the demonstration of the technology on the Experimental Aircraft project and finally leading to the collaboration with the Eurofighter Partner Companies (EPC's), building on European Software experience to specify, procure and release the EFA Software Development Environment (EFA SDE).

The paper goes onto describe those issues that are arising within the EF forum that could influence the development of SDE's for future military aircraft projects.

This paper is written from the viewpoint of British Aerospace and does not necessarily reflect the views of the other Eurofighter companies.

2 LIST OF SYMBOLS.

AB Allocated Baseline

BAe British Aerospace

CDR Critical Design Review

CORE COntrolled Requirements Expression

DoD Department Of Defence

EAP Experimental Aircraft Programme

EF Eurofighter

EFA European Fighter Aircraft

EF SDE Eurofighter Software Development Environment

EPC Eurofighter Partner Companies

EPOS Engineering And Project Management Oriented Support System

FB Functional Baseline

FCA Functional Configuration Audit

HOOD Hierarchical Object Oriented Design

IPSE Integrated Project Support Environment

LDRA Liverpool Data Research Associates

MASCOT Modular Approach to Software Construction and Test

MDS Microprocessor Development System

PB Product Baseline

PCA Physical Configuration Audit

PDR Preliminary Design Review

PVL Programme Validation Limited

SAFRA Semi-Automated Functional Requirements Analysis

SDE Software Development Environment

SSR Software Specification Review

STD Standard

TRR Test Readiness Review

VMS VAX Management System

3 PREPARING FOR EFA.

At British Aerospace the investigations into the requirements of the SDE for EFA began in 1978. At that time it was recognized that if EFA was implemented at the software productivity levels being found on TORNADO, Software Development effort would be excessive [8.1].

If EFA was to be affordable software productivity had to improve.

3.1 PRE EAP.

Looking ahead to EFA it was clear that a *step* change in productivity was necessary. Throughout industry and the academic world effort was being applied to tackle this problem. Although recognising this and in fact heavily leaning on the results of this activity BAe could not afford to wait and hope that this would solve the problem. BAe therefore embarked on a series of risk reduction exercises targeted at demonstrating the required productivity.

To begin with a software development process model was established following the now traditional waterfall approach. This model formed the basis of SAFRA (Semi-Automated Functional Requirements Analysis). The principle underpining SAFRA is *RIGHT FIRST TIME* achieved through early detection of errors. To achieve the

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

required improvement in productivity every activity must add value and all non-value added activities must be eliminated.

The SAFRA lifecycle is shown in figure 1.

The methods adopted on SAFRA were COntrolled Requirements Expression (CORE), Modular Approach to Software Construction And Test (MASCOT) and Enhanced PASCAL was adopted as the project language. The supporting toolsets were the CORE Workstation and PERSPECTIVE. In addition a micro-Processor Development System (MDS) was used to support low level, real time software testing.

The principles of each SAFRA phase can be summarised in figure 2.

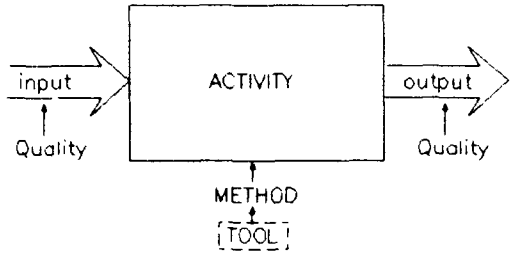


FIGURE 2: SAFRA Concepts.

- Inputs for each activity should be subject to strict quality control to ensure a stable baseline.
- The activity must be supported by a defined method to ensure consistency and repeatability.
- Each method should be supported by a tool to ensure quality and productivity.
- Each output should be subject to strict quality control to ensure a stable baseline for the next activity in the lifecycle.

The concepts, methods and tools of the SAFRA model were tried out on several study projects. Sufficient confidence in the SAFRA approach was established to enable it to be adopted on the P110 combat aircraft project (precursor to EFA) and finally as the development approach on the Experimental Aircraft Programme (EAP).

3.2 EAP.

The results of the application of SAFRA on EAP. [8.1] were very promising and showed that by adopting controlled methods, procedures and tools the productivity required for EFA could be achieved.

In addition, a significant improvement in the quality of the final product over that of previous projects was achieved. The approach had been able to detect errors earlier in the development lifecycle than on previous projects.

3.3 POST EAP.

Just prior to the completion of the EAP programme the requirements for the European Fighter Aircraft began to be defined.

It has been recognized by the EFA Customer that in addition to technical and performance requirements there is a real need to minimise the cost of ownership of EFA. To this end a strategy was formed to ensure that throughout its service life EFA will be easy to maintain and modify.

Part of this strategy centred around the development of software. It was perceived that by controlling the software development process, significant savings could be made in the long term. The main principles behind the software development strategy are:-

- the uniform application of a common software development process,
- the adoption of a project standard programming language,
- the use of a standard processor throughout the architecture,
- the use of a common toolset for software development.

4 THE EFA SDE.

These principles outlined above were expanded in the EFA Development Contract to establish the requirements for the EFA SDE.

4.1 CRITICAL REQUIREMENTS.

The EF Software development process is required to be common throughout the project. The Development Contract cited DoD-std-2167 Software Development [8.2] as the required development model.

In order to define, specify procure and release the EF software Methods Procedures and tools a multi-National Software Management Group consisting of representatives from each EF Partner Company was established at British Aerospace, Warton.

The first task of this group was to tailor the DoD standards to accommodate the experience that had been gained from projects like EAP.

The team set about defining a set of standards covering Software Development, Software Configuration Management and Software Quality Evaluation.

An overview of the EF software development process is shown in figure 3.

The development process has to cater for differing software criticality levels encompassing both safety critical and mission critical software.

The EFA system and software is incrementally developed with increasing functionality and clearance over time.

The experience from each EPC on software development methods was used to define the EF Method set. The methods chosen to support the EFA lifecycle were:-

- COntrolled Requirements Expression (CORE) for requirements capture and system design,
- Hierarchical Object Oriented Design (HOOD) for software design,
- Ada and a safe Ada subset, for safety critical software, as the programming languages,
- and the use of Host computer and Target emulation for software testing.

The benefits arose from building on EAP experience (CORE and Target emulation), adopting industry standards (Ada) and tracking leading edge expertise (HOOD). The next challenge facing the Software Management team was to specify and procure a set of tools to support these methods and to move from a program support to a project support environment.

4.2 SELECTION PROCESS.

The fundamental requirements driving the EF Toolset selection were :-

- Each tool must be compatible with VAX VMS and run either on a VAX Terminal or a VAX Station.
- The tool must support EF methods, process or the development and testing of Ada programs.
- Each tool will drive productivity and quality into the EF Process.

The Software Management team performed a survey of the software tool market to establish the state of the art.

As a result of this survey, a set of tool specifications were written defining the required functionality and performance of the EF SDE. For the majority of the tools a competitive tendering exercise was conducted with tenders received from several of the major software tool suppliers in Europe. Each offered tool was technically evaluated against a checklist of requirements by the Software Management team. A separate commercial tendering exercise was handled by EF Procurement.

The final outcome of the tendering exercise resulted in the selection of the EF SDE. The toolset of the EF SDE is described below.

4.3 THE EF SDE TOOLSET.

The EF Software tools are EPOS, CORE Workstation, HOOD Toolset, EFA Ada Compilation System, SPARK Analyser, LDRA TESTBED and the EF IPSE. It is not the purpose of this paper to describe in detail the EF toolset however, a brief overview of each tool is provided for information.

4.3.1 EPOS.

EPOS (Engineering Project management Oriented support System) is a tool developed by GPP mbH in Germany. It is used on EFA to structure English requirements documents into a more modular, traceable form. It enables each requirement contained in such documents to be allocated a unique requirements identifier.

Eg (Note: The following fictitious examples are provided to illustrate the technique).

Requirement 10431(4233).

The aircraft will be capable of releasing its payload at all altitudes within the flight envelope.

Requirement 20423(1454).

The aircraft will be capable of performing its mission in all weathers.

This number is then used throughout the remainder of the development as a reference for compliance traceability. The EPOS tool automatically supports a compliance check listing where requirements have been fulfilled and highlighting any unfulfilled or partly fulfilled requirements.

4.3.2 CORE Workstation.

The CORE Workstation is a VAX Station based tool produced by British Aerospace to support the CORE method. The tool consists of a diagram and text editor, report generator and database checker. The tool also supports individual diagram and text note configuration control.

4.3.3 HOOD Tool.

The HOOD Toolset is a VAX Station based tool produced by IPSYS Ltd. The HOOD tool consists of a diagram and text editor, report generator and database checker. It also has a facility for automatic code generation (Ada). The tool follows the method outlined in the HOOD Reference Manual [8.3].

4.3.4 Ada Compilation System.

The EFA Ada Compilation System is a VAX VMS based toolset comprising:-

EDS SCICONS's

- XD Ada Target compiler,
- XD Ada MC68020 HP Emulator interface

DIGITALS's

- VAX/VMS Ada Host compiler,
- Language Sensitive Editor,
- Source Code Analyser

Performance Coverage Analyser,

Softspeed's

MC68020 simulator

The complete toolset is procured from EDS SCICON. The Ada Compiler is used throughout the EFA project and is managed by a combined Customer/Contractor management group. This group controls the baseline compiler and plans for changes to that baseline.

4.3.5 SPARK Examiner.

Safety critical software for EFA must be written to a (Safe) Ada subset which restricts the use of Ada to a deterministic set of features. This subset is enforced through compliance with the SPARK Ada subset and checked using the SPARK Examiner tool.

The SPARK Examiner (Version A) is produced by PVL Ltd. It is used on EFA to statically analyse safety critical software. It checks conformance with the SPARK Ada language subset and performs control flow, data flow and information flow analysis.

4.3.6 LDRA Testbed.

The TESTBED tool is produced by the Liverpool Data Research Associates Ltd. It is used on EFA for both static and dynamic analysis of EFA code. It determines code metrics such as McCabe Cyclomatic complexity, it provides test effectiveness ratios for statement, branch and Linear Code Sequence and Jump coverage. It also performs Data Set analysis mapping test cases to code analysed.

4.3.7 EF Integrated Project Support Environment (IPSE).

The EF IPSE is based on the PERSPECTIVE KERNEL produced by EDS SCICON. The EF IPSE performs automated configuration management of the products produced during the system and software development lifecycle, it enables the other tools to be integrated into the IPSE allowing full control of the configuration of tool products automatically. It also allows for the controlled definition, allocation, development and return of specific packages of software development.

Whereas EAP utilised an environment for support of software design, code and test, on EFA a full IPSE is necessary in order to control a much larger product and still achieve productivity levels similar to those on EAP.

The IPSE structure is represented in the figure 4.

The concept is that all software development work is carried out within the IPSE. This will enable every package of work to be configured from its inception.

The IPSE supports a geographically distributed database allowing packages to be tracked as they are sent throughout Europe. This will support large, distributed team working.

Currently HOOD and Ada are integrated into the IPSE. Other tools can be integrated when necessary through use of the IPSE developers kit.

The IPSE supports the EF configuration management procedures and forms and therefore allows automated control of change and tracking of configuration status.

4.4 CURRENT STATUS OF EF SDE.

The EF SDE has been fully released to project and is being used for the development of the current EFA development software. The effectiveness of the toolset has yet to be determined and will only become fully clear at the end of development.

5 LOOKING TO THE FUTURE.

A number of issues have arisen during the development of EFA software. It may be possible to accommodate them within the timescales of EFA but this will be based on a case by case cost/benefit analysis.

The root of these problems which are highlighted by these issues, centres around the fact that for the past 15 to 20 years the discipline of software engineering has been very heavily biased towards tool development. Development of processes and methods has not matched the speed of tool development.

If we are to make any real progress in the future, effort has to switch from developing faster and more capable tools to developing repeatable, measurable development processes that integrate into an overall aircraft development programme.

We must put focus on the products that we are developing rather than on the tools to support them.

I have described some of the particular issues arising from the EF forum below.

5.1 SYSTEMS NOT JUST SOFTWARE.

The overall process of developing a complex military aircraft involves many requirements being addressed by many disciplines over a very short period of time. (In fact, it is increasing likely that the development time and cost must be reduced below current levels).

Each of the different disciplines (Software design, installation management, aerodynamics, operational analysis, flight test, airframe design and manufacture etc) must integrate together at key points during the development of the aircraft.

Software engineering up to now has tried to address a perceived problem with the specification, design, code and test of *software*. Most development models mention the existence of other disciplines, however the integration aspects are sadly overlooked. In software we have focused on getting *software* productivity up and *software* quality right.

The resultant SDE's have allowed us to produce bigger, better dataflow diagrams, compile larger amounts of code faster and test individual lines of code more completely than ever before.

But what about the system?

How do we ensure that the information needed to produce the aircraft wiring drawings is provided in the timescales required to the drawing office instead of when required for software development?

How do we cater for hardware lead times which require implementation information upfront when we are following a top down lifecycle model which postpones such details until just prior to software design.

How do we ensure that the aerodynamic models being used by Software designers within the SDE are consistent with those used by airframe engineers and aerodynamics in their respective toolsets?

As software becomes a more important element of the overall System we are in danger of taking longer to develop the software than to build an airframe. We may even begin to delay the production of the airframe if we are unable to provide the necessary integration details in airframe development timescales.

For future aircraft projects, the software development model needs to be expanded up to include its complete integration with other development models. The selection of the appropriate methods and tools can then be made to ensure that the required information is developed in time for the overall aircraft project, not just for software.

5.2 INCREASING PRODUCTIVITY.

It is highly likely that the next generation of combat aircraft will have considerably more software than EFA. It is also likely in a highly competitive market place that the timescales from development through to production will be required to be less than that of EFA. Cost constraints will mean that we will have to develop this software with relatively fewer engineers.

All these pressures add up to a need for another *step change* in productivity.

Effort for SDE development can focus in this area on concepts such as reuse, rapid prototyping and automatic code generation from requirements. Note that these are changes to the development models not just efficiency improvements within individual tools.

5.3 INCREASING DOCUMENTATION.

The current lifecycle approaches are also likely to swamp the project under a forest of documentation. There is a real need for the next aircraft to be almost totally paperless, if not for management reasons then purely for ecological reasons.

Again the emphasis should switch from tool development to method improvement. Particularly in the area of requirement abstraction. We need to change the saying "A picture paints a thousand words" to "A dataflow diagram automatically creates 1,000 lines of optomised code".

Process Improvement documentation must add value not just satisfy standards.

5.4 INCREASING RELIANCE ON SOFTWARE.

As the use of software spreads across all applications on aircraft there will be an increasing reliance on the safety of software.

It takes considerable effort to ensure that software errors will not have hazardous consequences. Methods and tools to support Formal specification analysis and proof of software have to be developed such that they are effective and practical.

It must be ensured that these techniques when introduced do not affect productivity so dramatically that we can never afford to develop a new aircraft system.

5.5 IMPROVING MAINTAINABILITY.

A key factor to ensuring that future projects are affordable is by reducing considerably the cost of ownership. Not just ensuring that there are few or no errors in the product but also ensuring that there is sufficient growth built in to accommodate the design changes that will happen throughout the service life of the aircraft.

Current practice involves mandated project wide languages (Eg Ada), developing around standard architectures, building in processor and memory growth capabilities and using modern modular, top down design methods.

Adopting much more modular, reconfigurable Avionic architectures could significantly increase the adaptability of future weapon systems and would also support reuseability.

5.6 CONTIGUOUS METHODS.

Modern software engineering methods and tools have been developed to try to eliminate ambiguities in specification and design which would result in errors in the code. A whole plethora of different methods and approaches have been developed all claiming to be the best at solving the problems of particular parts of the software development lifecycle.

It is almost unavoidable to have to choose different methods across the lifecycle. This introduces the potential for translation error and discontinuity when changing from one method to another.

Integrated compatible methods built on a well understood process model will not only provide a smooth, error reduced path through the lifecycle but will also provide a basis for automatic generation of lifecycle products including code.

5.7 STABLE TOOLSETS.

Once an aircraft project has developed and cleared a large part of its software, it will be reluctant to have to rewrite or reclear this software as a result of a software toolset and host operating system change. The most obvious tool which can cause such an effect is the compiler.

The project must make a trade off between tracking the technology changes triggered by changes in the software tool market and the need for stability during the development programme and on into maintenance.

Again this is an area where the development of software tools is constraining the very projects that they have been developed to support.

5.8 MULTINATIONAL COLLABORATION.

Future projects are very likely to be based around some form of multinational collaboration. Companies in Europe are each investing in particular software development strategies based on the current method and tool market place. These strategies will set the nature of that company's tool procurement and staff development programmes.

When a collaborative project begins the partners must be able to work together choose an integrated SDE possibly compromising significantly on their chosen strategy and causing a significant amount of new tool procurement and staff training.

If there was an industry standard SDE adopted by the majority of European aerospace companies or a framework for interchange then this would facilitate much closer collaboration on software development.

5.9 METRICS.

It is a fundamental quality aim to have a repeatable, well measured, development process. This enables sound estimation, visibility and provides targets for sensible process improvement.

However in multinational projects each company as well as being a collaborator in a particular project is possibly a competitor in other projects. This makes it extremely difficult to share sensitive metric information (Eg productivity and quality levels). However these very metrics are essential to be able to effectively manage the project.

A metrication system established for such projects must either be based around agreed commercially, protective contracts or by a process of gathering the metrics without being able to attribute them to any particular company. These metrics could also be published to enable the method and tool developers to understand the real problems of software development.

6 CONCLUSIONS.

The EFA SDE marks a significant step change in productivity from that of previous BAe projects. A similar step change will be necessary to manage the next generation of military combat aircraft.

In the future software development has to be considered as a process within an overall framework of Aircraft development. Software methods and tools should be developed to fully support this.

The issues raised in this paper are being addressed in forums across Europe and the United States however, progress has to be made to ensure that the needs of the next large military aircraft programme are fully satisfied prior to the start of its development.

7 ACKNOWLEDGEMENTS

Thanks to A.Bradley, D.Beck, B.Corcoran, C.J.Everingham, A.Matthews E.Sefton and A.Williams for their help in checking this paper.

8 REFERENCES

8.1 A.O.Ward., "Software Engineering: Another Small Step", BAe-WAA-R-RES-SWE-314

8.2 Department Of Defence., "Software Development", DoD-STD-2167

8.3 ESA., "HOOD Reference Manual"

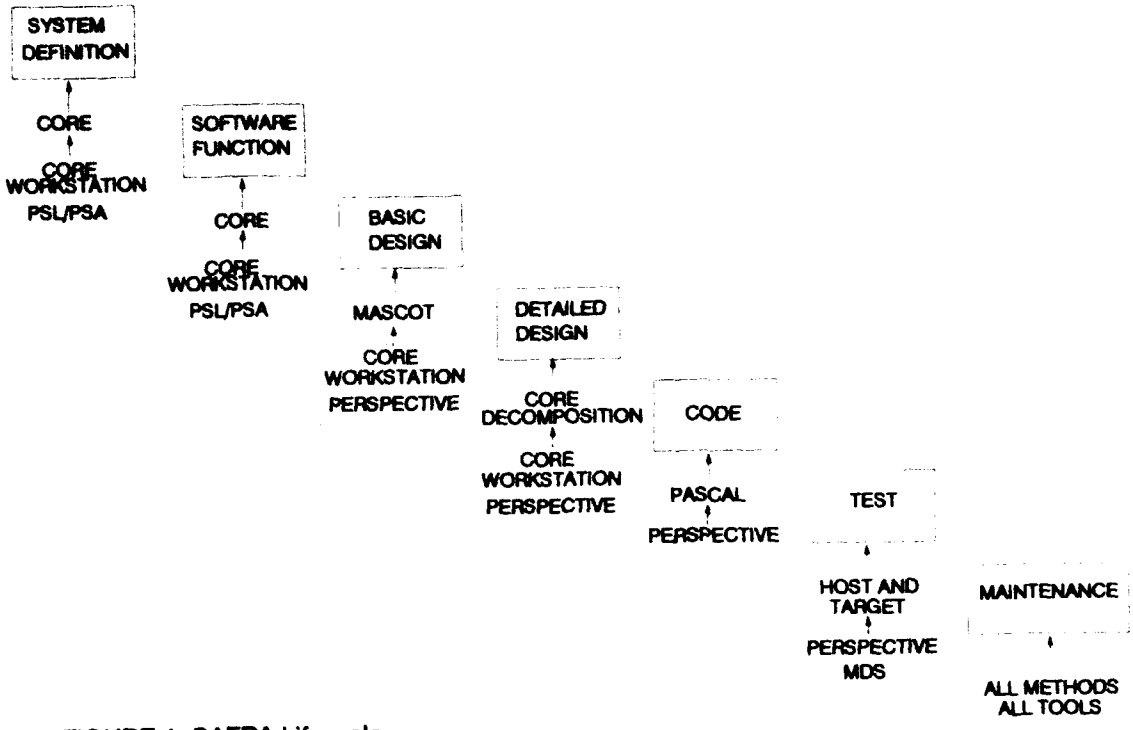


FIGURE 1: SAFRA Lifecycle.

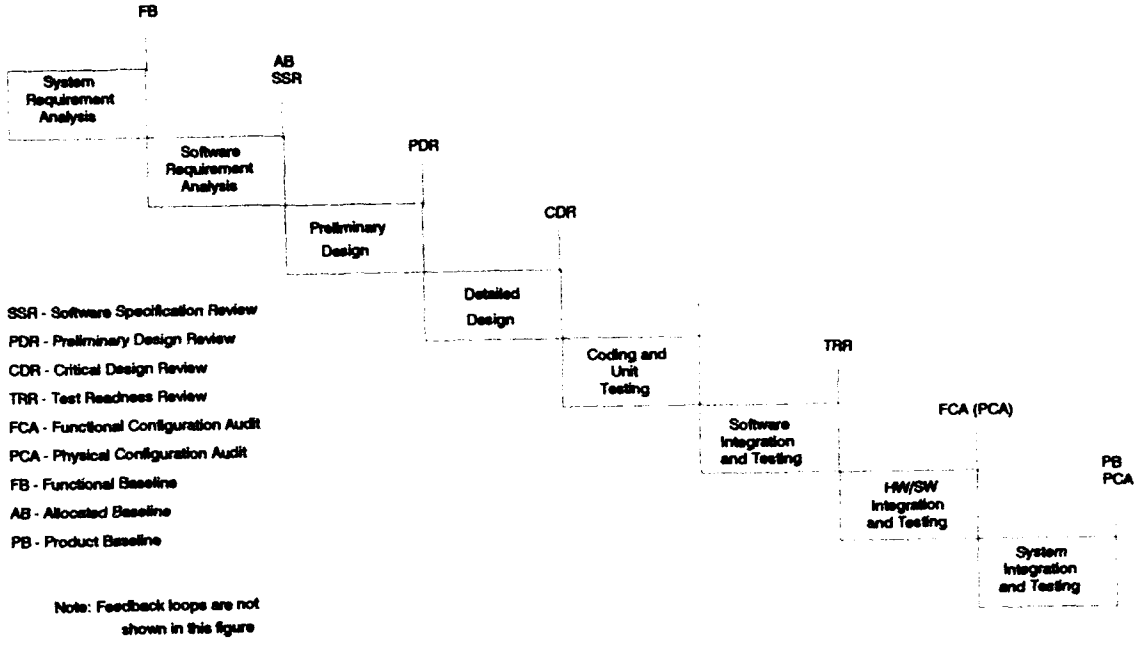


Figure 3: EFA Software Development Lifecycle.

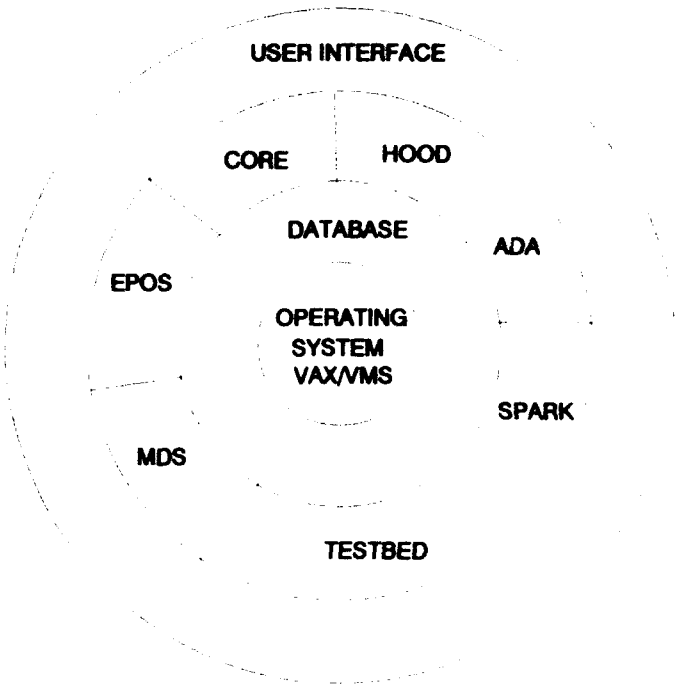


Figure 4. EF IPSE STRUCTURE.

Discussion

Question W. ROYCE

Can you support contiguous operation in which object-oriented software building is done side-by-side with procedural (i.e. functional) software building? What are the problems?

Reply

In having a mixed method approach, which is feasible, the main problem is of traceability accross method boundaries. For instance, data is represented centrally in OOD but distributed in functional decomposition. In following such an approach, a lot of effort will be expended mapping from one structure to another.

A contiguous method would eliminate this problem. Currently, contiguous methods do not appear to be being researched seriously.

Question D. NAIRN

Are you not barking up the wrong tree in focusing on the tools? Tools are a means to an end. If you focus on an engineering description, then your information is not being driven by the tools/host computer. (There is no fundamental reason to have more than one type of database, and more than one graphics editor, etc, in the entire environment).

Reply

I agree that the notation of the design should be independent of the tools. Unfortunately, this was not the case during EFA SDE selection. We use CORE/HOOD and are tied to those tools. However, the problem I refered to in the paper is mainly concerned with changes to target computers occurring during development which may result in unnecessary retest due to the effect of the compiler change on code when re-compiling from existing library's.

To change now from our current toolset to a new generic notation is possible, but may be too costly and take too long in the development programme.

Un environnement de programmation d'applications distribuées et tolérantes aux pannes sur une architecture parallèle reconfigurable

Ch. FRABOUL, P.SIRON
C.E.R.T.-O.N.E.R.A.
Département d'Etudes et de Recherches en Informatique
2 Avenue Ed Belin BP 4025 31055
TOULOUSE CEDEX FRANCE

1. SOMMAIRE

Les travaux menés dans le cadre du projet MODULOR se sont intéressés à la spécification et la mise en oeuvre d'une machine massivement parallèle modulaire et reconfigurable dynamiquement, ainsi qu'à la mise en œuvre des outils logiciels nécessaires pour l'exploitation des possibilités de reconfiguration lors du développement d'une application parallèle. La reconfiguration de l'architecture a été étudiée tout d'abord sous l'aspect fonctionnel qui vise une adaptation automatique de la topologie d'interconnexion pour une mise en oeuvre la plus efficace possible d'une application donnée. L'apport d'une capacité de reconfiguration de la structure d'interconnexion pour assurer la tolérance de l'architecture parallèle aux pannes des processeurs a été abordé dans un second temps.

Ces travaux mettent en évidence la complémentarité des deux types de reconfiguration ("fonctionnelle" et "en cas de panne") pour définir une architecture assurant un maximum d'efficacité de communication, dans un environnement temps réel qui nécessite de pallier la défaillance de certains des éléments de la machine.

2. INTRODUCTION

Les besoins de traitement des systèmes informatiques du futur (en particulier au niveau du traitement des signaux radars, contre-mesures, sonars....) nécessitent la définition de machines massivement parallèles. Les contraintes de connexion d'un nombre important de processeurs de traitement (plusieurs centaines), conduisent à des architectures du type réseau de processeurs dont la structure d'interconnexion conditionne les performances. De plus, les problèmes de tolérance aux pannes, en particulier des processeurs de traitement, imposent la définition d'une structure d'interconnexion permettant la connexion de ressources de secours .

Dans une architecture du type réseau de processeurs chacun des processeurs ne dispose que d'un niveau de mémoire locale mais peut communiquer en mode message avec les processeurs auxquels il est relié par des liens de communication. Une telle architecture est dite reconfigurable dynamiquement lorsque la topologie d'interconnexion des processeurs peut évoluer, au cours de l'exécution d'une application, en fonction des commandes fournies à la structure d'interconnexion. Les architectures à haut degré de parallélisme reconfigurables présentent des caractéristiques intéressantes : limitation du nombre de connexions par processeur, connexion directe des processeurs communicants (en évitant des mécanismes de routage), adaptation de la topologie à un problème donné, prise en compte de fonctionnement en mode dégradé, transparence et souplesse de l'architecture.

Cependant ce concept de reconfiguration de la structure d'interconnexion d'une architecture du type réseau de processeurs a été jusqu'à présent utilisé de deux manières relativement indépendantes :

- d'une part la reconfiguration fonctionnelle qui vise essentiellement l'adaptation (la plus automatique et la plus dynamique possible) de la topologie d'interconnexion à un problème (ou un sous-problème) donné [1].
- d'autre part la reconfiguration en cas de panne qui a principalement comme objectif d'isoler un élément défaillant de l'architecture pour permettre la poursuite des traitements en cours (éventuellement en mode dégradé). Cette forme de reconfiguration a été plus particulièrement étudiée dans le cas des architectures du type tableau de processeurs [2],[3].

La spécification et la mise en oeuvre d'une machine massivement parallèle reconfigurable dynamiquement se heurte à plusieurs problèmes :

- au niveau architectural, il s'agit de prendre en compte les contraintes technologiques telles que la taille des commutateurs utilisables pour construire la structure d'interconnexion reconfigurable, le nombre de liens de communication disponibles par processeur, les moyens de commande de la structure d'interconnexion.
- au niveau logiciel, le problème est d'offrir à l'utilisateur les outils permettant l'exploitation de la reconfiguration de la topologie d'interconnexion aussi bien pour des raisons d'efficacité que de tolérance aux pannes.

3. ARCHITECTURE RECONFIGURABLE

La connexion de l'ensemble des liens de communication de tous les processeurs d'une architecture parallèle sur un commutateur unique est rarement possible dès que le nombre de processeurs devient important (même dans le cas de liens de communication séries). Malgré les évolutions technologiques prévisibles, se pose alors le problème de définition d'un réseau d'interconnexion dit à

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

étages qui présente l'inconvénient d'un temps de transfert plus important. La solution que nous avons retenue consiste à bénéficier au maximum de la localité des communications et s'appuie sur une structure d'interconnexion modulaire.

Une première idée consiste à associer un commutateur à chaque type de liens des processeurs (Nord, Sud, Est, Ouest, dans le cas d'un processeur disposant de quatre liens de communication série). Cette hypothèse permet de diviser la complexité de chacun des commutateurs par le nombre de liens de communication d'un processeur. Elle conduit donc à la définition d'un module de l'architecture qui permet de connecter localement un nombre de processeurs égal au maximum à la taille des commutateurs (comme représenté figure 1).

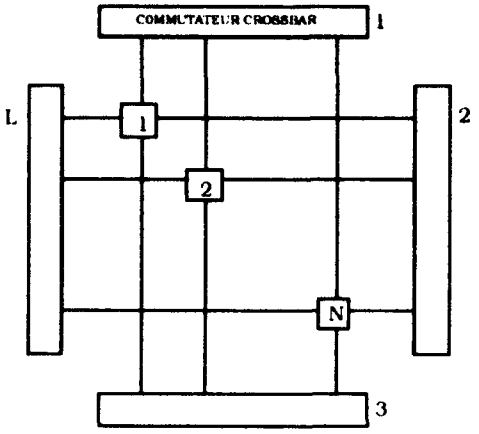


Figure 1 : architecture d'un module

Une seconde idée consiste à définir, de manière récursive, un réseau permettant l'interconnexion de tels modules (comme schématisé figure 2). Un certain nombre de liens de communication est alors réservé sur chaque réseau interne à un module pour la communication inter-modules.

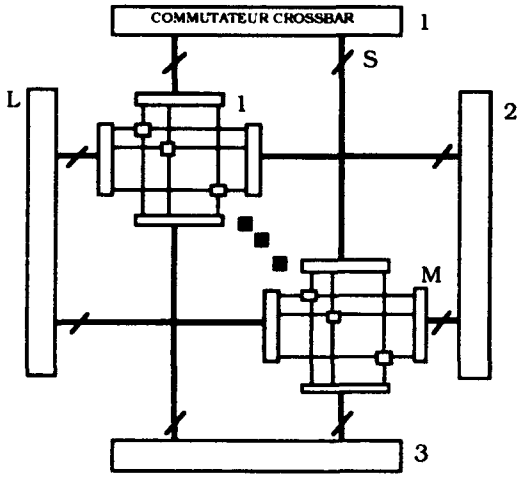


Figure 2 : architecture multi-modules

L'architecture ainsi définie est une architecture modulaire, composée d'un certain nombre de modules reliés entre eux par une structure d'interconnexion reconfigurable, chaque module est constitué d'un ensemble de processeurs totalement connectés par une structure interne également reconfigurable. Un lien de communication entre deux processeurs est donc établi via le réseau intra-module uniquement si les deux processeurs sont situés sur le même module. Le réseau inter-modules n'est nécessaire que dans le cas où les deux processeurs communicants sont situés sur deux modules différents [4].

Cette architecture peut être caractérisée par les quatre paramètres suivants:

- M : nombre de modules,
- N : nombre de processeurs par module,
- L : nombre de liens par processeurs,
- S : nombre de liens de chaque commutateur interne réservé pour la communication inter-module.

Cependant plusieurs modes de connexion des différents commutateurs internes et externes à un module sont envisageables. Nous verrons plus loin que la conception des structures d'interconnexion intra et inter-modules a été validée par la démonstration de la capacité de l'architecture ainsi définie à supporter une application reconfigurable.

4. APPLICATION RECONFIGURABLE

La mise en oeuvre d'une application sur une telle architecture parallèle reconfigurable nécessite des outils spécifiques si l'on veut utiliser pleinement les possibilités de reconfiguration dynamique de la topologie d'interconnexion. Nous nous intéresserons dans un premier temps à l'aspect reconfiguration fonctionnelle, c'est à dire à la recherche d'une (ou plusieurs) topologie(s) d'interconnexion adaptée(s) aux besoins de communication de l'application. Nous supposerons que le découpage de l'application en modules parallèles a déjà été effectué (comme pour une mise en œuvre sur une architecture à topologie d'interconnexion fixe).

Il semble actuellement irréaliste de vouloir déterminer automatiquement les instants où la topologie d'interconnexion devrait être modifiée. L'hypothèse retenue consiste à décrire une application parallèle reconfigurable comme une succession de phases algorithmiques, pouvant présenter des besoins de communication différents. Ces phases seront séparées par des points de reconfiguration explicitement introduits par le programmeur. Ce dernier décrira également l'enchaînement souhaité de ces différentes phases (qui peut dépendre des résultats de traitements).

Une phase algorithmique donnée peut être alors décrite sans connaissance de l'architecture sous-jacente. Nous supposerons que l'application peut être représentée sous forme de processus communiquant en mode message (en utilisant, par exemple, un modèle de programmation concurrente de type CSP : "Communicating Sequential Processes" [10]). On appelera processus l'unité d'allocation de traitements sur un processeur de

l'architecture (ce qui n'exclut pas une granularité plus fine de processus exécutés en quasi-parallélisme sur un processeur). Chaque phase peut donc être représentée par un graphe où un nœud représente un processus et un arc (orienté) un lien de communication logique reliant deux processus (comme montré figure 3).

Le problème de développement d'une application reconfigurable porte donc sur l'analyse des différents graphes de communication de l'application et sur la détermination automatique de la topologie la mieux adaptée à chacune des phases de cette application. Nous avons fait le choix d'un interface graphique pour la description des différents graphes de communication de l'application.

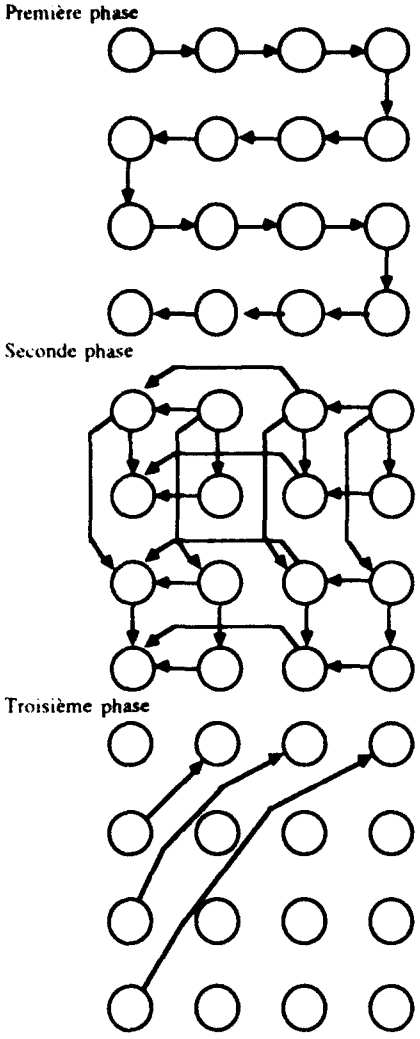


Figure 3 : graphes de communication d'une application

Les outils de développement qui ont été définis comportent principalement trois modules [4]:

- un interface graphique permettant la description d'une application reconfigurable externe de graphes de processus communiquant en mode message,
- un outil de placement de graphes de processus sur un réseau de processeurs reconfigurable qui conduit à la détermination de la topologie d'interconnexion nécessaire et des commandes correspondantes des différents commutateurs,
- un logiciel de génération de code qui assure les mécanismes de synchronisation nécessaires avant tout changement de la topologie d'interconnexion. Le code généré pour cette synchronisation se situe au niveau des processeurs, mais également au niveau de la machine hôte qui gère l'envoi des commandes aux commutateurs pour réaliser les changements de topologie et l'enchaînement des différentes phases.

5. PLACEMENT D'UNE APPLICATION RECONFIGURABLE

Les outils de développement qui viennent d'être présentés incluent une étape importante de placement des graphes de processus associés au différentes phases de l'application reconfigurable sur l'architecture multi-modules proposée. La modularité de cette architecture induit en effet des problèmes supplémentaires dus à la limitation du nombre de liens de communication inter-modules.

Pour simplifier l'exposé, nous supposerons que le nombre de processus de chaque phase est inférieur ou égal au nombre de processeurs de l'architecture et que le degré de connectivité de chaque processus est inférieur ou egal au nombre de liens physiques disponibles sur chacun des processeurs. Ces hypothèses sur le graphe initial occultent deux problèmes complexes que nous ne développerons pas :

- le premier est celui de la contraction d'un graphe dont le nombre de processus dépasse le nombre de processeurs disponibles. Ce problème de contraction qui vise le regroupement de processus sur le même processeur est équivalent au problème de partitionnement que nous évoquerons plus loin [8],
- le second est celui de la transformation d'un graphe de processus dont le degré de connectivité est supérieur au nombre de liens disponibles sur un processeur. Ce problème peut nécessiter la création de processus de multiplexage permettant de gérer le partage d'un même lien physique par plusieurs liens logiques.

Un algorithme de placement d'une application distribuée a pour objectif d'allouer les processus qui constituent cette application sur les processeurs disponibles en assurant les communications nécessaires. Compte tenu des hypothèses qui viennent d'être rappelées et du fait que tous les processeurs sont banalisés, le problème de placement du graphe de processus associé à chaque phase sur un tel réseau de processeurs reconfigurable se ramène à la détermination des liens de communication qui doivent être établis entre processeurs en fonction des processus qu'ils exécutent [9]. Après analyse du graphe de communication, il s'agit bien de déterminer la topologie d'interconnexion adaptée à chaque phase de l'application.

Le placement d'un graphe de processus sur l'architecture multi-modules définie pose deux problèmes [6] :

agit d'une part de partitionner le graphe global en sous-graphes faiblement interconnectés de telle manière que les contraintes de nombre de modules et de connexion entre modules puissent être satisfaites (voir figure 4),
- il faut assurer d'autre part que chacun des sous-graphes obtenus puisse être alloué sur un module de l'architecture, c'est à dire que les liaisons intra et inter-modules puissent être établies (comme représenté figure 5).

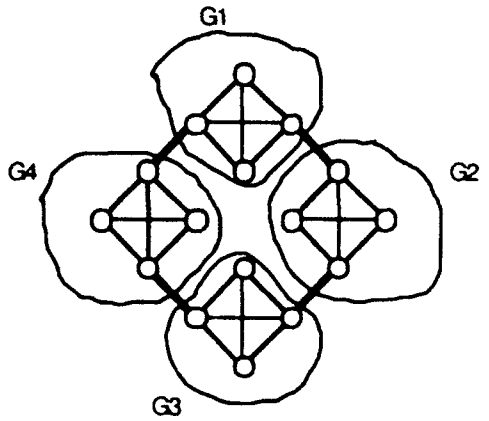


Figure 4 : graphe partitionné

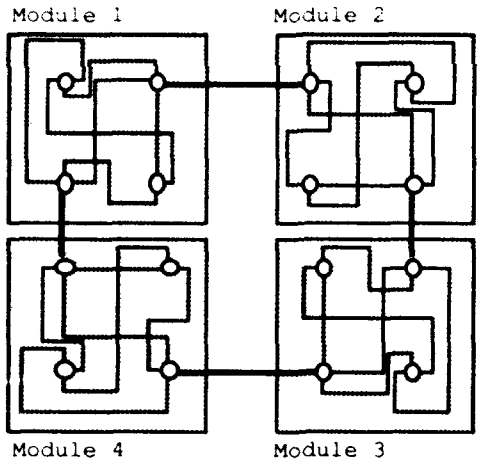


Figure 5 : placement du graphe partitionné
(N=4, M=4, L=4, S=1)

Des heuristiques de partitionnement de graphes ont donc été développées. Elles s'appuient sur une notion d'affinité entre processus qui permet de permet de constituer le nombre de partitions souhaité en minimisant les communications entre partitions (prise en compte de la contrainte des liens de communication externes à un module L*S). Ces heuristiques ont été validées sur des graphes fortement connectés (hypertores..) et sont d'une complexité polynomiale inférieure à celle d'autres heuristiques connues.

D'autre part la conception des deux réseaux d'interconnexion a été guidée par la démonstration de la capacité de l'architecture ainsi définie à supporter le placement de toute application codée en termes de processus communiquant qui soit partitionnable.

Ainsi, le placement d'un graphe de processus sur un module de l'architecture impose une hypothèse supplémentaire qui consiste à associer deux à deux les commutateurs d'un module. Dans le cas d'un processeur à 4 liens, on disposera d'un réseau Nord/Sud et d'un réseau Est/Ouest (si le lien Out du lien bi-directionnel est connecté au réseau Nord,le lien In est connecté au réseau Sud, comme représenté figure 6).

De même, le placement de tout graphe partitionnable n'est garanti que si le réseau inter-modules est constitué de telle manière qu'un commutateur externe supporte les liens externes d'un rang donné dans chacune des directions de connexion provenant de tous les modules (comme représenté sur la figure 6).

Compte tenu de ces deux hypothèses architecturales, il a été démontré qu'il est possible de placer tout graphe de processus partitionnable vérifiant les propriétés énoncées plus haut sur l'architecture multi-modules [6].

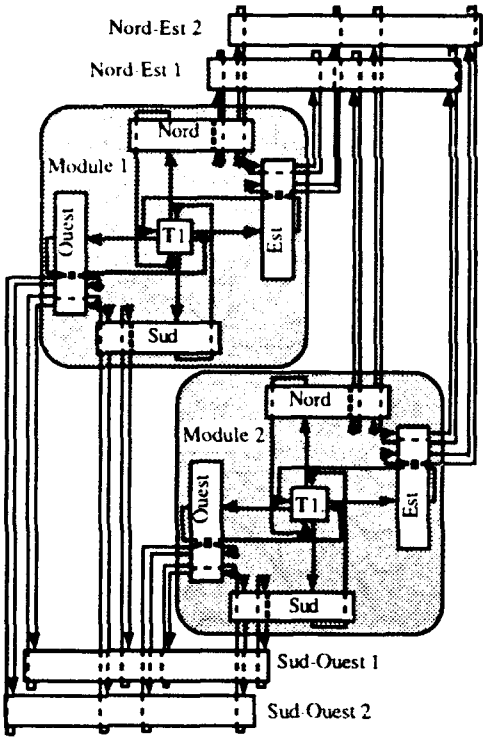


Figure 6 : communications intra et inter-modules

Les deux niveaux de réseaux d'interconnexion peuvent mis en oeuvre à l'aide de commutateurs élémentaires de taille raisonnable [5] :
- un réseau intra-module est composé de L commutateurs internes reliant un type des L liens de communications des N processeurs d'un module. Sur

chacun des L commutateurs internes d'un module S ports sont réservés pour la communication inter-modules. Un commutateur interne doit donc pouvoir commuter, au minimum, N+S ports d'entrée vers N+S ports de sortie.
- le réseau inter-module retenu est composé de 2*S commutateurs qui relient chacun L/2 liens de même rang provenant des M modules. Chacun de ces commutateurs doit donc pouvoir commuter M*L/2 ports d'entrée vers M*L/2 ports de sortie.

6. RECONFIGURATION ET TOLÉRANCE AUX PANNES

Pour augmenter la disponibilité du système, nous nous intéresserons surtout aux techniques de résolution de pannes matérielles, pannes auxquelles on peut parfois assimiler des pannes logicielles (par exemple les pannes qui se traduisent par un silence du processeur défaillant).

Une architecture parallèle peut assurer une redondance statique et le masquage d'erreur [3]. Cette technique déjà éprouvée pour les traitements séquentiels est souvent utilisée dans les systèmes embarqués. La réplication des traitements et des mécanismes de vote majoritaire sont généralement mis en œuvre. Le modèle TMR ("Triple Modular Redundant") est un exemple de la classe des méthodes de programmation en N versions. Ces techniques intègrent des mécanismes sophistiqués pour traiter les problèmes de communication y compris dans le processus de vote (protocoles d'agrément byzantin).

Les mécanismes de redondance dynamique, peuvent être mis en œuvre sur une architecture reconfigurable. Ils permettent également de garantir la continuité des services si des cycles de reprise sont admis. Les principes sont la détection d'erreur, les techniques permettant de repérer l'élément défectueux, les mécanismes permettant la reprise .

La redondance dynamique matérielle qui est visée repose donc sur la possibilité de reconfiguration de la topologie d'interconnexion permettant le remplacement de processeurs défaillants par des processeurs de secours. La connexion de processeurs additionnels sur chaque module de l'architecture, permet de remplacer n'importe quel processeur défaillant d'un module. La redondance matérielle peut également être prise en compte au niveau d'un module (connexion d'un module additionnel). Nous négligerons dans un premier temps les pannes des réseaux d'interconnexion ou plus exactement nous supposerons que les moyens de communication peuvent être redoublés [7].

Compte tenu des capacités de reconfiguration de l'architecture, la mise en œuvre de techniques de redondance dynamique peut être réalisée par logiciel. Les mécanismes de détection de pannes reposent sur la notion de phase. Une phase correspond à l'exécution d'un graphe de processus communicants et son exécution est automatiquement précédée par un algorithme de synchronisation de début de phase et suivie par un algorithme de synchronisation de fin de phase. Si nous supposons que la panne d'un processeur lors de l'exécution d'une phase conduit à un interblocage (hypothèse du processeur silencieux en cas de panne). Il est possible de démonter que cet interblocage se produira quelque soit l'instant de la panne: début de phase, code algorithmique, fin de phase. En particulier les échanges nécessaires pour la synchronisation de fin de phase conduiront au blocage même si les processus de la phase algorithmique ne communiquent pas.

Les mécanismes mis en place permettent successivement :
- de détecter le blocage des processus pendant l'exécution de la phase,
- de débloquer les processus en attente sur un ou plusieurs liens de communication,
- d'attendre la terminaison des processus bloqués (synchronisation de fin de phase minimale) avant d'enchaîner avec la phase de diagnostic.

Les mécanismes de détection du blocage des processus (généré par une communication qui ne se termine pas), reposent sur l'introduction d'un délai d'exécution maximal pour chaque phase. Le calcul de ce délai peut être réalisé à l'aide de mesures d'exécutions des différentes phases dans un environnement exempt de pannes. Le code des mécanismes de détection du blocage repose d'une part sur l'utilisation des horloges et de mécanismes de détection d'anomalies de transmission. Des mécanismes de ré-initialisation des canaux de communication sont également nécessaires.

La phase de diagnostic est gérée comme une phase additionnelle. Au cours de cette phase, le superviseur, c'est à dire la machine hôte établira un dialogue avec les différents processeurs. La panne d'un lien de communication sera considérée comme la panne totale du processeur, car il est a priori nécessaire de disposer d'une connectivité totale pour assurer l'exécution du code usager. Cette phase de diagnostic met à jour des tables d'état du système, utiles pour l'exécution des phases suivantes, mais également pour la maintenance.

La reconfiguration de l'architecture consiste à remplacer le processeur défaillant par un processeur de secours. le nombre de processeurs de secours détermine le nombre de pannes pouvant être prise en compte lors de l'exécution d'une application. Le principe de mise en œuvre est celui de la reconfiguration fonctionnelle dans la mesure où tous les processeurs (de traitement et de secours) sont des processeurs banalisés :
- le processeur de secours, déjà possesseur du code de l'application reçoit l'identité logique de la tâche à assurer,
- les commandes du réseau interne au module sont appliquées de telle manière que le processeur de secours vienne remplacer le processeur défaillant dans la topologie,
- les commandes des réseaux externes sont conservées.

Il est important de noter que tout partitionnement et placement dynamiques sont écartés. Il n'est pas envisagé de fonctionnement en mode dégradé sur un nombre restreint de processeurs. La notion classique de point de reprise, s'applique pour reprendre le traitement de

l'application en cours. Une ré-initialisation du contexte au début de la phase courante permet de confondre point de reprise et point de synchronisation de début de phase. Cette stratégie suppose qu'une phase de sauvegarde du contexte des processus soit insérée après chaque terminaison normale de phase. Cette sauvegarde distribuée est effectuée par un système de voisinage et est bien sûr elle même protégée contre les pannes. La phase, symétrique, de restauration des contextes est mise en œuvre, avant la ré-exécution de toute phase interrompue.
L'environnement de développement décrit précédemment a été modifié pour mettre les mécanismes assurant la tolérance aux pannes de processeur. Les modifications apportées à la chaîne de développement d'applications reconfigurables sont relativement minimes. L'usager doit annoter chaque phase d'une durée d'exécution maximale. la structuration de l'application en phase peut cependant être revue pour optimiser le déroulement de l'exécution du programme en cas de panne. Il peut être en effet souhaitable de rajouter des points de synchronisation (de reprise) dans le cas d'une phase trop longue pour obtenir un ensemble de phases de grain plus fin.

Le code correspondant aux mécanismes de tolérance aux pannes est inséré automatiquement par le générateur de code.

7. MAQUETTAGE ET VALIDATION

Une maquette fonctionnelle de l'architecture spécifiée a été réalisée à partir de composants standard INMOS : le Transputer T800 qui dispose de 4 liens de communication et le commutateur crossbar C004 qui permet de connecter 32 voies d'entrée sur 32 voies de sortie. Les valeurs des paramètres retenues pour ce maquettage sont : M=4, N=20, L=4, S=8.

L'architecture permet de bénéficier de la localité de connexion des liens de communication des processeurs au niveau d'un module et de minimiser le nombre de connexions inter-modules L'intégration d'une telle architecture peut donc être réalisée de manière relativement simple :

- des cartes mères assurent pour chacun des modules l'implémentation du réseau intra-module,
- des cartes filles sur lesquelles se trouvent les processeurs et leur mémoire viennent s'enficher sur les cartes mères,
- une carte fond de panier réalise l'interconnexion des cartes mères en implémentant le réseau inter-modules.

Cette maquette a permis une validation en vraie grandeur de l'architecture multi-modules et de la chaîne de développement d'applications reconfigurables. Ces outils, écrits en langage C, utilisent pour la partie graphique la bibliothèque GMR2D disponible sur les stations de travail HP-Apollo. Ils sont en cours de portage sous l'environnement XWindow. Les codes des processus de l'usager, reliés grâce à l'interface graphique pour former une seule application, sont actuellement écrits en langage OCCAM [11].

Les premières applications mises en oeuvre sur cette maquette ont montré le gain apporté par l'utilisation des possibilités de reconfiguration de la topologie d'interconnexion. Les performances sont augmentées de près d'un tiers pour certaines applications, par rapport à leur implémentation sur la même architecture configurée selon une topologie fixe (grille). Ces résultats dépendent cependant de nombreux paramètres : matériels utilisés d'une part (temps de reconfiguration, coût de routage...), caractéristiques de l'application d'autre part (volumes de transferts, distance et variation de communications entre processus...).

Cet environnement de développement, étendu pour l'aspect tolérance aux pannes, a été validé sur quelques applications. Les pannes des processeurs ont été simulées par l'ajout de points d'arrêt dans les codes de certains processus. Le coût des mécanismes assurant la tolérance aux pannes n'a pu faire l'objet de mesures fines. Il est actuellement directement lié aux mécanismes de sauvegarde qui n'ont pas fait l'objet d'optimisations particulières.

CONCLUSION

L'étude a permis de mettre en évidence l'apport d'une structure d'interconnexion totalement reconfigurable permettant de gérer :

- une reconfiguration fonctionnelle de la topologie d'interconnexion évitant au maximum les mécanismes de routage par une connexion directe des processeurs communiquant entre eux,
- une reconfiguration en cas de panne visant le masquage de la défaillance d'un ou plusieurs processeurs rendue possible dans la mesure où tous les processeurs sont banalisés et que des processeurs de secours sont également reliés à cette structure d'interconnexion.

Les travaux présentés ont montré l'apport d'une architecture modulaire pour la mise en œuvre d'une structure d'interconnexion reconfigurable. Cette modularité est également utile pour la prise en compte de mécanisme de tolérance au pannes.

Cette étude a de plus démontré la faisabilité de solutions pouvant être apportées au niveau logiciel pour permettre, en fonction des hypothèses architecturales, de poursuivre l'exécution d'une application après détection d'une anomalie et reconfiguration de l'architecture.

D'autres approches seraient à envisager pour assurer la tolérance aux pannes d'architectures du type réseau de processeurs en particulier celles consistant à allier des méthodes de type masquage d'erreur avec des méthodes du type détection de panne [12].

Les travaux menés dans le cadre du projet MODULOR ont fait l'objet de contrats DRET (Direction des Etudes Recherches et Techniques de la DGA) et sont soutenus par le MRT (PRC Architectures Nouvelles de Machines) et la région Midi-Pyrénées.

RÉFÉRENCES

1. K. Hwang, Z.Xu, "Remps: a reconfigurable multiprocessor for scientific supercomputing", Proceedings Parallel Processing 1985
2. M.Chean, J.Fortes, "A taxonomy of reconfiguration techniques for fault-tolerant processor arrays", Computer, january 1990
3. V.Nicola, A.Goyal, "Limits of parallelism in fault-tolerant multiprocessors", 2nd IFIP Int Conference DCCA 1991
4. V.David, Ch.Fraboul, J.Y.Rousselot, P.Siron, "Etude et réalisation d'une architecture modulaire et reconfigurable", Rapport DRET n°1/3364, Mars 1991
5. V.David, Ch.Fraboul, J.Y.Rousselot, P.Siron, "Définition d'une architecture modulaire et reconfigurable", 3ème symposium PRC ANM Palaiseau Juin 1991
6. V.David, Ch.Fraboul, J.Y.Rousselot, P.Siron, "Partitionning and mapping communication graphs on a modular parallel architecture", CONPAR-VAPP5, Lyon, septembre1992
7. Ch.Fraboul, P.Siron, "Etude d'une architecture reconfigurable tolérante aux pannes", Rapport DRET n° 2/3420/DERI, novembre 1992
8. P.A.Nelson, L.Snyder, "Programming solutions to the algorithm contraction problem", Proceedings Parallel Processing 86, 1986.
9. E.K.Lloyd, D.A.Nicole, "Switching Networks for Transputer Links", Proc. of the 8th OCCAM Users Group, Technical Meeting, mars 1988
10. C.A.R.Hoare, "CSP: Communicating Sequential Processes", Prenctice Hall 1985
11. D.Poutain, D.May, "A tutorial introduction, to OCCAM programming" INMOS doc. 1988
12. F.Cristian "Understanding fault-tolerant distributed system", Communication of the ACM, february 91

Discussion

Question W. MALA

1. In case of reconfiguration, how will the software package be loaded to the spare processor under real time conditions?

2. How can you ensure, in case of a failure, which data are still valid and which are wrong?

3. What amount of time will be required for reconfiguration?

Reply

1. Le logiciel est actuellement chargé statiquement sur les processeurs. Il s'agit d'un logiciel générique(identique pour tous les processeurs). Le code déroulé par un processeur dépend de son numéro d'identification. Les processeurs de secours disposent de ce code et reprennent les numéros des processeurs défaillants.

2. Les données sont sauvegardées en fin de phase (notion de sauvegarde de contexte). Dans le cas d'une défaillance d'un processeur, l'exécution de la phase courante est reprise à partir des données sauvegardées (à la fin de la phase précédente). Un processeur de secours doit pouvoir accéder aux données qui avaient été sauvegardées par le processeur qui vient de tomber en panne. Des mécanismes de duplication des sauvegardes sur des processeurs voisins sont mis en œuvre.

3. Temps de commande des réseaux d'interconnexion (avec la technologie utilisée) ≈ 10 µs.
Temps de reconfiguration, incluant les mécanismes de synchronisation par échange de messages (avec les processeurs T 800 et des liens de communication à 10 Mbits/s) ≈ 1,5 ms.

A COMMON APPROACH FOR AN AEROSPACE SOFTWARE ENVIRONMENT

F.D. Cheratzu
Alenia - Finmeccanica S.p.A.
Corso Marche, 41
10146 Torino (ITALY)

SUMMARY

AIMS is a European industrial research project which focuses on the process for the development and maintenance of Embedded Computing Systems which are an integral part of high technology aerospace products. It is a user driven project which uses a problem oriented approach to solve the difficulties encountered in the production of such systems. The relevance of the proposed solutions to the problems is ensured by involving aerospace engineers, who work on the development and maintenance of embedded systems. This involvement ensures that new technologies, or improved practices, can be rapidly introduced into operational projects.

LIST OF SYMBOLS

AIMS Aerospace Intelligent Management and development environment for embedded Systems
ALN Alenia
AS Aerospatiale
A340 Airbus 340
BAe British Aerospace
EC Eurocopter
ECS Embedded Computing System
EFA European Fighter Aircraft

1 INTRODUCTION

The AIMS Project is a unique Project within the EUREKA programme which addresses the problems companies have in developing and maintaining the complex embedded systems found within many aerospace products. The AIMS Project is looking to improve the development and maintenance process for embedded systems to maintain the competitive advantage the European Aerospace industry has achieved through collaborative projects. Its area of application has been recognised by the EUREKA initiative as being of great importance to the future success of the European aerospace industry.

The aim of this paper is to present an overall view of the AIMS Project and a description of the technical approach that has been adopted for the current phase of the Project.

The background to AIMS is described along with the organisation of the Project. The Project's overall objectives and strategy are described, which provide the context for the current phase. A summary of the technical approach is provided with an indication of the type of technical work which is being undertaken. The conclusion describes the results we have achieved to date.

2 BACKGROUND TO THE AIMS PROJECT

High technology products, such as aircraft, spacecraft, helicopters and missiles contain increasingly complex Embedded Systems like flight control, avionic and cockpit systems. The trend within these systems is to develop Embedded Computing Systems (ECSs) that provide significantly more functionality without the weight and size penalties of traditional electro-mechanical systems. One of the consequences of this trend is the rapid growth of the software within ECSs as is illustrated in figure 1. The ECSs now account for at least one third of the overall cost of the high technology aerospace products and have a significant impact on the timescale for developing these products.

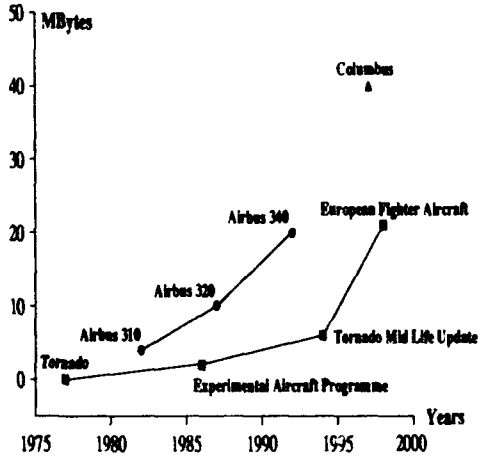


Fig. 1: Growth of on-board SW

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

Market forces and political pressures have compelled the aerospace companies to collaborate in a large number of international programmes. Through these collaborative initiatives such as Airbus, Ariane and Tornado, the European aerospace industry has obtained a competitive advantage in the world market. This success has led to an increasing number of collaborative initiatives such as EFA 2000 (European Fighter Aircraft), ATR (regional transport planes), Columbus (orbiting space station) and Tiger (combat helicopter). It is envisaged that the vast majority of all future aerospace projects will be collaborative.

It is a major challenge for the European aerospace industry working in collaborative projects to develop the complex ECSs on time and within budget. The need in the future for even more complex systems will require more effective collaboration to share the high development costs.

Within this context, three major European aerospace companies, from now on called Partner Companies, Aerospatiale (France), Alenia (Italy) and British Aerospace (United Kingdom), decided to cooperate through a research project called AIMS.

The Partner Companies are engaged in all phases of design, production and maintenance of a large variety of sophisticated aerospace products and have substantial experience in the production of the Embedded Computing Systems installed within these products.

3 PROJECT OBJECTIVES AND STRATEGY

The overall objective of the Project is to:

> *Reduce the cost of collaboratively developing and maintaining ECSs by enhancing productivity, stabilising timescales and improving cooperation, while ensuring the required quality levels are maintained.*

Therefore the focus of AIMS is on improving the ECS development and maintenance process and not on any particular ECS product. Neither is it targeted to one specific aerospace project but it is intended to bring long term benefits to a large number of future projects.

The AIMS Project mission is to obtain agreement with future collaborative partners on the required improvements to the ECS development and maintenance process to ensure future projects can develop complex ECSs on time and within budget.

To support this mission a strategy has been developed which recognises the trends within the aerospace industry:

- the majority of future projects will be collaborative;
- an increasing number of ECSs will be used to provide the increased functionality required for future systems;
- an increasing proportion of the development and maintenance costs of the aerospace products are due to the ECSs.

The strategy then builds on the strengths of the participating companies and on their particular needs.

The strategy recognises that the current ECS development process and its current use of methods and their supporting tools do not fully satisfy the requirements of the aerospace companies and will be unable to support the production of future aerospace systems. For the environments to meet the challenges presented by the rapid growth of embedded systems, new ways of working supported by new technologies must be found to enhance their productivity.

AIMS believes that it can play an important role in improving the process of ECS development and maintenance; not by developing a unique environment for developing all the future aerospace ECSs, but by:

- improving and harmonising the ECS development and maintenance process;
- industrialising new emerging technologies;
- defining common requirements based on real problems for the type of tools and support environment required by the European aerospace industry and
- influencing emerging standards which will have an impact on the support of the ECS development process.

This strategy is being implemented in the following way:

1) The AIMS Project must first look at how the aerospace companies work on ECS development now and in the near future, to identify the problems they experience with their working practices and the available technologies. Differences and commonalities of the various ECS development processes have to be identified and analysed to prepare the convergence towards the AIMS generic ECS development process.
2) AIMS must then indicate potential solutions to the problems identified above, which could be new techniques or technologies, or the improvement of existing techniques available from vendors.
3) The potential solutions must then be assessed to prove their industrial viability for solving aerospace problems. The assessment work will be distributed among the Partner Companies, not only to share costs but also to involve

practitioners inside the companies and ensure the assessments are based on real case studies. This will ensure that results can be used immediately and thus gain short term benefits.

4) The solutions which have been recognised as enhancing working practices through agreed criteria will be kept and integrated together. The AIMS generic ECS development process will be modified in order to support these solutions allowing their immediate use by the Partner Companies, thus gaining medium term benefits.

5) Finally, based on an agreed AIMS ECS development process supporting new industrialised solutions to actual problems, the AIMS team will be able to make strong recommendations to vendors concerning the environments, the tools and the methods which could be used to support the development process, thus gaining long term benefits. Through this cooperative work the AIMS Project will be in a strong position to influence future emerging standards.

To carry out this strategy, the AIMS partners have chosen to employ the majority of the Team within the aerospace industry in order to benefit from a detailed knowledge of aerospace practices without being tied into the production deadlines of any specific product. However, when additional help is required, the AIMS Team draws on the experience of many other experts from national research laboratories and system or software houses.

The AIMS Team is confident that the implementation of the strategy outlined above will result in helping the European aerospace industry to work together with more efficiency when developing and maintaining ECSs, thus retaining its competitive advantage.

4 PROJECT PHASING AND ORGANIZATION

The Project received EUREKA status in September 1987. Preliminary discussions were conducted between the original five Partner Companies[1] to establish the long-term objectives and the overall Project organization.

During the *Definition Phase* an investigation was undertaken to determine how the Partner Companies carry out the ECS development and maintenance process and the common problems experienced by them. The potential solutions for these problems were then studied theoretically for various phases of the development process such as specification, design and testing.

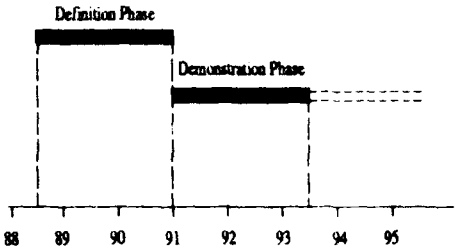


Fig. 2: AIMS Project Phasing

During the *Demonstration Phase*, the solutions identified in the definition phase are being implemented and, using real case studies, their industrial applicability is being evaluated. The solutions in isolation are not sufficient, therefore research into the integration of the solutions is being undertaken.

In the next phase, the potential integration technology is to be assessed and, using real case studies, their industrial capability shall be evaluated. A migration strategy to converge aerospace projects towards the AIMS process improvements shall be initiated.

AIMS is a user-driven project with its roots well inside the participating companies. To maintain strong links with the companies, members of the Team work within their own organisations and therefore the AIMS Team is distributed between the countries of the three Partner Companies. Efficient communication between all parts of the team is vital, to allow a wide exchange of information, and this need has led to a well defined but flexible organisation which forms the backbone of the Project.

A clearly defined hierarchy of groups co-ordinates, controls, monitors and executes the Project work. This structure, which follows the EUREKA project organisation guide-lines, facilitates democratic decision making and helps to ensure that each partner company actively supports all the Project decisions.

5 TECHNICAL APPROACH FOR THE DEMONSTRATION PHASE

The Demonstration Phase has the objective of providing evidence of the benefits that may be gained by implementing the solutions proposed in the Definition Phase, in order to reduce the risks of developing or acquiring new technologies. For this purpose industrial demonstrators have been set up to evaluate and exploit, where possible, techniques and technologies available on the market and, in some cases, to develop new techniques.

1 CASA (Spain) and MBB (Germany) participated in the Project from 1988 to 1991.

5.1 Industrial Demonstration

AIMS has developed a common approach for the definition and assessment of the demonstrators to ensure that their results are applicable to all the Partner Companies.

This approach requires that both the problem and solution are fully understood. This understanding can then be used in the definition of criteria to assess the solutions, so that the assessment will provide acceptable proof that the solutions will have real benefits over current techniques.

To understand the problems in the ECS development and maintenance process it was necessary to consult aerospace practitioners who have first hand experience of these problems. By fully understanding the concepts underlying the proposed solutions it is possible to identify the impact that these solutions would have on the efficiency of the ECS development and maintenance process.

Having identified the expected impact of the solutions, it is possible to define the scope and criteria for assessment of the solutions in order to provide acceptable evidence of the benefits.

It is important that the solutions are assessed using real project information in order to show the viability of the solutions in the real world. Proof that the solutions provide real benefits will be obtained by comparing the results of assessments of the new solutions with those obtained from using current techniques. By involving the aerospace practitioners closely at all levels of the demonstrations the results will be immediately available to them for use in improving their ECS development and maintenance processes on current projects, even before an overall result for AIMS is achieved.

The four AIMS demonstrators are briefly described below:

1) *Collaborative Working in Systems Development* - This demonstrator is being undertaken by Eurocopter France. Its objective is to investigate problems currently encountered when developing systems collaboratively, especially when the participating partners are geographically distributed. This demonstrator is assessing various collaborative working techniques which will improve the sharing and communication of project information, and the required organisational support required by collaborative projects. The assessment of this demonstrator will be performed based on a sub-system of the Tiger helicopter.

2) *Prototyping and Animation of ECS Specifications* - This demonstrator is being undertaken by the Military Aircraft Division of British Aerospace Defence Ltd. Its primary objective is to investigate whether the use of prototyping and animation can aid in the validation of system and software specifications, and reduce errors introduced during the requirements and specification phases. The demonstrator will assess an improved notation for specifications which can also be animated. The assessment will be based on sub-systems from both Hawk and EFA Projects.

3) *Formal Methods for Software Design* - This demonstrator is being undertaken by the Avionics and Systems Direction of Aerospatiale's Civil Aircraft Division. Its objective is to assess whether the use of formal methods in the design of software for Embedded Computing Systems may help to reduce the cost and time required for software certification. The demonstrator will assess the use of a formal notation which can be formally refined to Ada code. The assessment will be based on a sub-system from the A340.

4) *Support for ECS software test activities* - This demonstrator is being undertaken by Alenia Defence Aircraft Division. Its primary objective is to explore innovative tools and techniques to support software testing of Embedded Computer Systems and to evaluate their impact on effort and quality. The demonstrator will assess: improved techniques for testing and the state-of-the-art tools used to support these techniques, an expert system to assist with the management of the testing activities, and the feasibility of generating test cases from formally defined specifications. The assessment of this demonstrator will be based on a sub-system from EFA.

Figure 3 shows the areas of the life cycle as covered by the demonstrators.

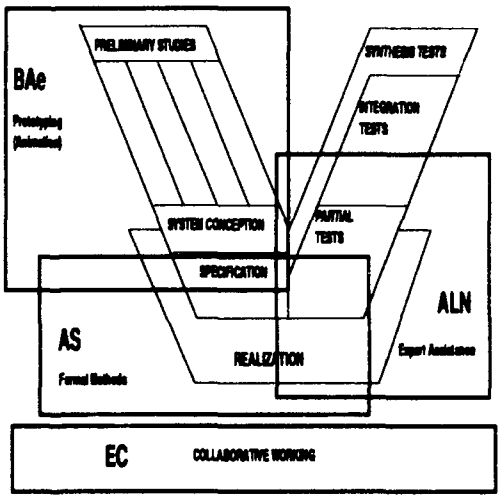


Fig. 3: The 4 Demonstrators

The involvement of the practitioners in the definition and assessment of the demonstrators has started the process of introducing the new techniques and technologies into the aerospace organisations, which is often one of the major reasons why new technologies are not adopted.

5.2 Integration

Through the demonstrator projects we are looking at improvements at local areas within the life cycle. To make use of all these improvements, we have to identify a means to integrate the different technologies. This is to be achieved by modelling the activities that are performed within the development process, the information produced and consumed, and the controls applied to the activities and the information. Collectively these models form the ECS Development Model and this will cover essential areas related to technical development, technical management and organizational management.

By using the models we can identify the data which is required and produced by each demonstrator. Based on this information we can determine how to interface between the demonstrators and provide a specification for their integration. The models can then be used as the means of communication between the Partner Companies as well as providing a more formal definition of our requirements for potential suppliers.

The models have been developed to show that integration may be achieved. This concept will be proven through the development of an integration demonstrator.

5.3 Exploitation

The aim of the exploitation work is to define how the Partner Companies can acquire the AIMS solution in a timely and cost effective way. This work will identify how environment related initiatives external to AIMS may be utilised, and how external bodies may be influenced to move towards the AIMS philosophy.

This will ensure that work being performed outside is not duplicated, but that it could be modified to meet the needs or long term requirements of the aerospace community.

5.4 Migration

The main reason for the existence of multi-aerospace collaborative Research & Development is to share its risk and cost and then to share the benefit of its results on commercial ventures in the future. As a consequence the results need to be transferred into use for the benefits to be gained.

Achieving this transfer requires approval of the R&D results by a company or collaborative project; and is not necessarily automatic. For example, the results of the ECS development process improvements will identify how much such improvements will save and how much they will cost (eg. in re-equipping and training development staff). Decisions on their transfer into industrial use have to be taken at a strategic level within the companies.

One solution is through the set-up of Migration Demonstrators which focus on the problems of transferring proposed process improvements onto aerospace development projects or into companies (based on process improvement demonstrators previously carried out in other companies). They would be conducted following the AIMS approach. The objective would be to re-use the problem analysis and proposed solution of a completed demonstrator assessment, in order to carry out a low cost company or project specific assessment. This would result in the confirmation or revision of the original process improvement results, but more importantly would achieve a wider acceptance of the results.

This is seen as a means of bringing about technology transfer at the collaborative level in the early years of the use of these process improvement techniques. In its own right, it forms a low cost, low risk alternative to (or supportive element of) the proposed migration programme.

6 CONCLUSIONS

This paper has provided an overview of the Project, and the pragmatic approach it has adopted, in order to achieve its goals. The following sub sections identify the results and influences the AIMS Project has had and foresees.

6.1 Significant Results Achieved to Date

To date, we have had some very significant achievements. Each of which has been agreed by the aerospace companies. These are:

- a common understanding of our objectives, expressed in terms of refined characteristics;
- a common understanding of the ECS development and maintenance process, expressed in terms of a process model - the AIMS ECS Development Model;
- an identification of the major aerospace problems, expressed in terms of the process and its impact on our objectives;
- an aerospace migration strategy.

However, these are only the paper documents that describe our achievements, but do not describe their impact, which is what really counts. We have seen a change in attitude within the companies, and a realisation that the approach we have defined has real benefits. This type of analysis of problems is spreading within our companies. AIMS has been seen as a model project for the level of cooperation it has achieved between the partner companies. It is seen as the way to resolve problems with potential future partners before we get to the project stage of aircraft and ECS development.

Finally, we have had an impact on other international initiatives such as the Portable Common Interface Set (PCIS) Programme, ensuring they take into account the users' view, which has resulted in the recognition, now widely accepted, that the user have an important role to play in the definition of standards.

6.2 Influence on Industrial Practice

AIMS has the advantage that it has access to a great wealth of industrial experience and industry practitioners. Therefore, it has tackled the problem of transferring improved working practices and support technology into industrial use in two complementary ways. The first approach was to understand what problems would prevent us from introducing any improved working practices and support technologies into the partner companies. The second was to use practitioners of real aerospace projects on the Demonstration Assessment projects.

The problems preventing technology transfer are of a political, financial and technical nature; all these have to be tackled if the Project is to be successful. The identification of these problems has resulted in the definition of a migration strategy, which identifies a pragmatic approach to introducing improved working practices and support technologies into the aerospace companies. This work has already influenced our work, in particular, in the way the demonstrator assessments are being performed.

The second approach of using practitioners of real aerospace projects is slightly more subtle. An intimate and pragmatic communication between practitioners and technology providers has been established. Solution providers are forced to understand the problems and working environment of the practitioners, rather than the practitioners having to understand the technologies and work out for themselves how it solves their problems. Finally the practitioners have been able to use the demonstrators and therefore have the confidence that the problems have been solved and that the solution is operationally applicable. This has resulted in the practitioners going back to their departments and selling the technologies to their colleagues and managers.

6.3 Significant Results Foreseen

In the short term we will receive the results from the demonstrator assessments which will identify how far we have gone towards achieving our goal of successfully introducing improvements in working practices and support technologies into the Partner Companies. The integration work will identify how improved working practices and support technologies can be integrated together. Relevant standards will be assessed to see if they are applicable in our domain.

This will enable us to identify the capabilities required for future aerospace projects using AIMS techniques before they are initiated, therefore leaving them to concentrate on getting the project work done.

In this way we believe we will be able to demonstrate the achievement of our objectives of: improved productivity, stabilized time schedules and effective cooperation.

Discussion

Question K. BRAMMER

You have mentionned that you proved the improvement, gained by applying the AIMS solutions, to decision-makers at decision level. This is a very important issue. Can you explain how you do this in practice?

Reply

Basicallyby giving the measures of the improvements. We use metrics from other projects, if they are available, otherwise we run case studies where measures are taken using previous techniques and later using the enhanced techniques. It is important to keep other conditions the same. This, coupled with the support of the practitioners, is to our experience the best way to convince a decision-maker.

Question C. BENJAMIN

What limitation did you run into when using Statemate?

Reply

People have some trouble initially with its notation, but this is quickly overcome.

A limitation has been encountered for the specification of real-time systems which have an intensive use of data.

A Distributed Object-Based Environment in Ada

M. J. Corbin
G. F. Butler
P. R. Birkett
D. F. Crush
Flight Systems
Defence Research Agency,
Farnborough, Hants
GU14 6TD, U.K.

1. SUMMARY

An object-based environment for implementing distributed systems is described. This can be used to create worlds of interacting objects, operating over a network of processors. The precise manner of the distribution is transparent to the objects within the environment.

A prototype of this environment is being implemented in Ada, augmented by support for object-oriented constructs. This is intended for use in real-time simulations of combat missions and will be known as Multi-sim.

2. INTRODUCTION

This paper describes some of the considerations behind the design of an object-based environment intended to support the implementation of certain types of distributed real-time system. The systems of interest for this work are typified by having a significant amount of global interaction among the various software entities represented within the environment, that is, ones in which it is not possible *a priori* to define localised limits for the interactions of any entity.

This characteristic is frequently met in combat mission simulators, in which a number of entities, or players, interact in various ways during the course of a simulated mission. It is not possible to say in advance which players will interact, or over what range the interaction will occur, and so all information governing such interactions needs to be globally available. Certain of these entities are controlled by human pilots, leading to a requirement for real-time operation, typically on a group of graphics workstations, linked by a medium-bandwidth local area network[1].

In attempting to design a software environment within which such distributed simulations can be conducted, the approach we have taken is to employ software objects to represent entities, groups of entities and component parts of entities.

The use of object-orientation in its full sense promises to result in system designs which are easier to maintain and enhance than previous approaches to software structure. The four main concepts usually regarded as characterising an object-oriented system are:

a) Encapsulation - a software object is completely self-contained, having all the code and data attributes it needs hidden within it, and accessible only through a well-defined set of operation calls.

b) Data Abstraction - the use of an object specification as a template to enable the creation of multiple instances sharing the same operations, but each with independent data attributes.

c) Inheritance - the ability to define a fresh object class having all the operations and attributes of an existing class, with additional attributes and operations of its own.

d) Dynamic Binding - the ability to specify an object operation to be performed, without needing to specify until run-time the class of object which will perform it.

It is important to emphasise that the use of object-oriented methods is not dependent on any particular programming language or environment. Rather it is an approach to organising and planning computer programs, an approach which can be applied to a greater or lesser extent in all software development. However, dedicated object-oriented programming systems such as Smalltalk 80[2], provide comprehensive support for the approach, and OO extensions to existing languages such as Objective C[3] or C++[4] have also been developed. The extent to which OO concepts can be realised in a Fortran environment has also been explored[5,6,7]. In addition, the features and data structures of Ada provide a good match to the requirements of OOD[8]. Within the DRA, work has concentrated on the provision of run-time support libraries for constructing worlds of interacting objects in Ada[9].

The Ada language was chosen for the main part of the environment because of its high degree of standardisation, portability and good software engineering features. It is not however, fully object-oriented as it currently lacks facilities for inheritance and dynamic binding. The environment makes use of Ada's encapsulation and data abstraction capabilities to enable the definition of self-contained classes of objects with well-defined interfaces. An emulation of Dynamic Binding is provided as a key part of the environment. This allows the core parts of the environment to have control over the operation of the objects within it, even though these objects may not exist when the environment is compiled.

In designing this environment, we have chosen to omit any direct use of inheritance, partly because it is difficult to implement in Ada, and also because we have found that defining component objects is a more flexible way of constructing complex objects for simulation purposes.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

For this reason, the term "Object-Based" has been used to describe the environment, in preference to "Object-Oriented". The design of component parts which can be readily re-used in different contexts is an important method for reducing the effort required to produce complex simulations.

One important constraint on this work was the requirement to be able to make use of a large set of existing models, mainly written in Fortran. This has been achieved by the provision for the use of customised Ada harnesses through which individual models can be controlled. The models themselves can then be written in any language, and are readily portable to other simulation environments[10].

3. THE DISTRIBUTED OBJECT DATA-BASE

The core part of this environment is a data-base containing basic information about all the objects in existence within the environment. The information stored includes the object's name, references to its owner and to its class and a list of component objects of which it is comprised. Each object fits into a hierarchy of component parts, starting with a single Top object.

The information in this data-base is replicated within each processor, so that each has a complete set of entries for all the objects in the other processors, as well as its own local objects. This replication ensures that access to the information in the data-base is fast, requiring no communication with other processors. When an object is created dynamically, an entry for it is made in the local processor's data-base, and a single message is broadcast to the other processors containing the information they need to create the corresponding entries within their own data-bases.

Objects can be destroyed dynamically, as well as created. Again, a single message suffices to update all the data-bases. The storage allocated to the object is retained within its class, so that it can be reused when another object is created. This eliminates problems caused by attempting to use the garbage collection facilities provided by various compilers.

The principle that each basic operation on the data-base results in only a single message between processors is very important for real-time operation of a multi-processor system. The alternative, in which an operation would involve a request message, followed by a response message, would result in complications within the requesting processor, which would either have to wait for the response, or remember to expect it on a subsequent cycle. The single message principle has been followed throughout this work, including the generic communication facilities described in the next section, and the simulation application described in section 5.

When applied to object creation, the single message principle means that a creation request made in one processor can return a reference to the new object immediately for use within the creating program, even though the new object may be an instance of a class implemented on another processor.

As well as supporting objects which are instances of a class, the data-base also has support for "single objects", which are not associated with any particular class. A single object is used to refer to a complete package of software which has not been written in the object-oriented style, and only contains a single set of attributes. This is particularly important when re-using software from other projects which has not been written using data abstraction. Creation and destruction of single objects is handled rather differently from creation and destruction of instances, since there is no class object to refer to.

The final facility offered by the object data-base is support for an emulation of dynamic binding. Ada does not currently permit dynamic binding, which involves selective calling of object operations, dependent on the type of object encountered at run-time. However it is vital to have this ability, since it permits the construction of general purpose facility packages which can make use of object operations without knowing at compile-time what classes of objects will be available to them. The simulation framework described in section 5 makes use of this principle to control the time integration of models, and to pass messages to them from other models.

4. GENERIC COMMUNICATIONS

The objects within this environment clearly need to communicate information governing the interactions between them. The environment has facilities to allow this communication to occur over a distributed world of objects, based on the following principles:

a) The nature of the information to be communicated is determined by the designer of the objects, not by the object environment. This ensures that the environment is truly general-purpose. This lack of specialisation is achieved by providing the communication facilities in the form of generic Ada packages, which are instantiated by the object designer to implement the specific communication requirements of the set of objects under consideration.

b) The communications are independent of the class of object being communicated with. It is frequently the case that identical information will be generated by (or required by) objects belonging to different classes. No distinction is drawn between these communications; in other words, it is not necessary to know what type of object is being communicated with either at compile time, or at run-time. This principle makes it possible to introduce new classes of object without redesigning, or even re-compiling, the existing classes, provided that the nature of the communication does not change.

c) The communications are also independent of the distribution of objects between the various processors in use for a particular job. The various routing operations required are handled transparently by the

environment as far as the objects are concerned. This is vital if objects are to be re-usable in different contexts. The same object code can be used for a non-real-time single processor work as for a real-time multi-processor simulation.

d) The communication packages can be added to in an incremental manner. This makes it possible to define fundamental communication services used by a wide variety of objects, while more specialised communications, used by a limited set of objects, can be added later, without affecting any of the other objects. Again, this encourages the re-use of existing object definitions.

The environment currently supports two distinct types of communication, one in which an object can request information about the state of another object, and a second in which one object can send a message to another. Both of these adhere to the principle, described above, that each basic operation within the environment be completed by a single inter-processor communication.

4.1 Data Stores

Data Stores provide the means by which one object can request information about the state of another. They provide for the global information transfer referred to at the start of Section 1. Data Stores behave like extensions to the object data-base; they have a slot for each object which can hold information about certain aspects of the state of that object and are replicated in each processor. When information is placed in the data store in one processor, it is automatically broadcast to all the others, and thus becomes global data available for inspection by any object in the system.

One useful feature of the data stores is that each one has an index to all the objects which have placed data in it. This can be used by an object retrieving the data to scan through all the data which is currently available within the environment, and thus explore the world of objects in which it finds itself.

This facility makes it possible to design objects which can interact with many other objects, without needing to be explicitly given the identities of those other objects. This greatly enhances the flexibility of use of objects within the environment and the ease with which the object population can be modified.

Data stores also contain information about the time latency or staleness of the data within them. This will allow implementation of extrapolation algorithms to minimise errors due to latency. These algorithms are not an inherent part of the environment, since the choice of whether or not to use them is one of the design trade-offs best left to the object constructor.

4.2 Event Handlers

In contrast to the data stores, in which the communication is initiated by the *receiver* of the information, an event handler allows the *source* of the information to initiate a communication. To do this, the source object schedules an event for the receiving object. This event, and any associated data relevant to it, is placed on an event queue in the event handler package. When this event comes to the head of the queue, the handler sends it on to the receiving object by forcing it to execute one of its operations.

Event handlers are instantiated by the object designer to handle sets of related events, each of which can have different data associated with it. They provide the means of constructing discrete-event simulation models, as described in the next section. Event handlers make use of the dynamic binding emulation facility to force objects to respond to their events. This ensures that the event handlers can be defined independently of the objects which will communicate through them.

5. APPLICATION TO SIMULATION

The main application currently envisaged for this multiprocessor environment is to real-time combat mission simulators. These comprise a number of "piloted workstations" - powerful graphics workstations equipped with a sub-set of aircraft controls - at which a pilot can command the operation of a single combat aircraft model within the simulation. A complete simulator comprises a number of such workstations, within which the aircraft models can interact with each other and with a variety of other models, such as missiles and ground forces. Combat mission simulators are used in DRA to investigate various aspects the design of mission support and weapon systems for aircraft under realistic conditions of simulated combat.

The environment described in the preceding sections will be used to implement a simulation support framework, Multi-sim, capable of running a simulator comprising multiple classes of models. Use of the generic communications mechanisms will allow the interactions between these models to be specified in ways which do not depend on the mix of other models in the simulation, or on the way in which they are distributed between processors. Figure 1 shows the overall software structure for the Multi-sim framework. Models can be written in a variety of computer languages, as long as each is provided with an Ada harness through which the environment can control the model. This feature is intended to encourage re-use of existing models written in C, Fortran or Pascal as well as development of new models written in specialised declarative languages like Prolog and Fril[11].

Both continuous-time and discrete-event models will be accommodated. Indeed, the same model can have both continuous and discrete aspects to its behaviour. The operation of the continuous models will be interleaved automatically with any discrete events so as to maintain them in time synchronisation. The control of both continuous models and discrete-events is to be performed by a scheduler, local to each processor. This will have both real-time and non-real-time modes of operation and will control the models in its processor by making use of the

dynamic binding emulation provided by the object data-base.

In order to do this, it will be necessary to generate Ada package bodies to call operations selectively from models at run-time, dependent on the type of model in use. These are referred to as "dynamic binding packages", and can be produced automatically by a code generator program. As they are the final pieces of software to be compiled, it will be particularly straightforward to introduce a new type of model into the simulation. All that is needed will be to modify the instructions to the code generator to include a reference to the specification of the new model, run the generator to regenerate the dynamic binding packages, and re-make the executable program. No other models or parts of the environment need be recompiled (Figure 2).

The communication packages required for the simulation are introduced by similar means. A group of models making use of a common set of communications packages can be formed up into a *model archive*, from which the models required for a specific simulation can be readily selected. These models should work together without needing any further modification. The development of archives of models for different purposes and levels of fidelity should greatly reduce the effort required to set up specific simulations.

The prototype simulation framework will be controlled initially by a temporary keyboard interface for interpreting the commands needed to create instances of models, clone them from existing instances (together with all their component parts), schedule events for them and run the simulation. All of these commands will obey the single message principle outlined above. This interface is to be constructed so that it can readily be replaced with more advanced Graphical User Interfaces when needed - neither the object environment nor the models need depend on it (Figure 2).

6. CONCLUSIONS AND FURTHER WORK

In this paper we have described an object-based environment intended for use on a multi-processing network having relatively low-bandwidth communications. Its main characteristics are:

a) It provides for a hierarchical decomposition of objects into component parts, with the component objects split between processors in an arbitrary manner.

b) It is written entirely in Ada, with provision for multi-language working within an object, to encourage re-use of existing code.

c) It provides generic communications between objects, both for objects to send information to others, and for objects to request information about others.

The main area of use envisaged for this environment is in the field of simulation, where a simulation framework, Multi-sim, supporting a mixture of pseudo-continuous and discrete-event modelling is being constructed. The initial target application is to the real-time combat mission simulations for both rotary-wing and fixed-wing aircraft, undertaken by DRA Farnborough. The archive of compatible models which will be built up for this purpose should also find use in hardware-in-the-loop testing of flightworthy equipment and also in operational effectiveness studies in related areas.

REFERENCES

1. Roden D. W., Harrhy D. A. "A Mission Adaptive Combat Environment (MACE) for Fixed and Rotary-Wing Mission Simulation", *in AIAA Conference, South Carolina*, August 1992.
2. Goldberg A., Robson D. *"Smalltalk-80: The Language and its Implementation"*. Addison Wesley, Reading(Mass), 1983 .
3. Cox B. J. *"Object Oriented Programming: An evolutionary approach"*, Addison Wesley, Reading(Mass), 1986.
4. Stroustrup B. *"The C++ Reference Manual"*, Addison Wesley, Reading(Mass), 1986.
5. Meyer B. *"Object Oriented Software Construction"*. Prentice Hall, New York, 1988.
6. Isner J.F. "A Fortran Programming Methodology based on Data Abstraction". *Communications of the ACM* 25, no. 10, p 686, 1982.
7. Corbin M. J., Butler G. F. "Object Oriented Simulation in Fortran", *in Society for Computer Simulation Eastern Multiconference*, Tampa, March 1989.
8. Booch G. *"Software Engineering with Ada"*. Benjamin/Cummings, Menlo Park, 1987.
9. Corbin M. J., Butler G. F. "A Toolkit for Object Oriented Simulation in Ada", *in Society for Computer Simulation Western Multiconference, Object Oriented Simulation*, pp 13-18, San Diego, January 1990.
10. Corbin M. J., Birkett P. R. "The Use of Object-Based techniques in a Multi-Lingual Simulation Framework", *in SCS European Simulation Symposium*, Dresden, November 1992, pp 203-207.
11. Monk R., Swabey M. "The Simulation of Aircrew Behaviour for Systems Integration using Knowledge-Based Programming", *in SCS European Simulation Multiconference*, Lyon, June 1993.

 This work was performed with the support of the Ministry of Defence.

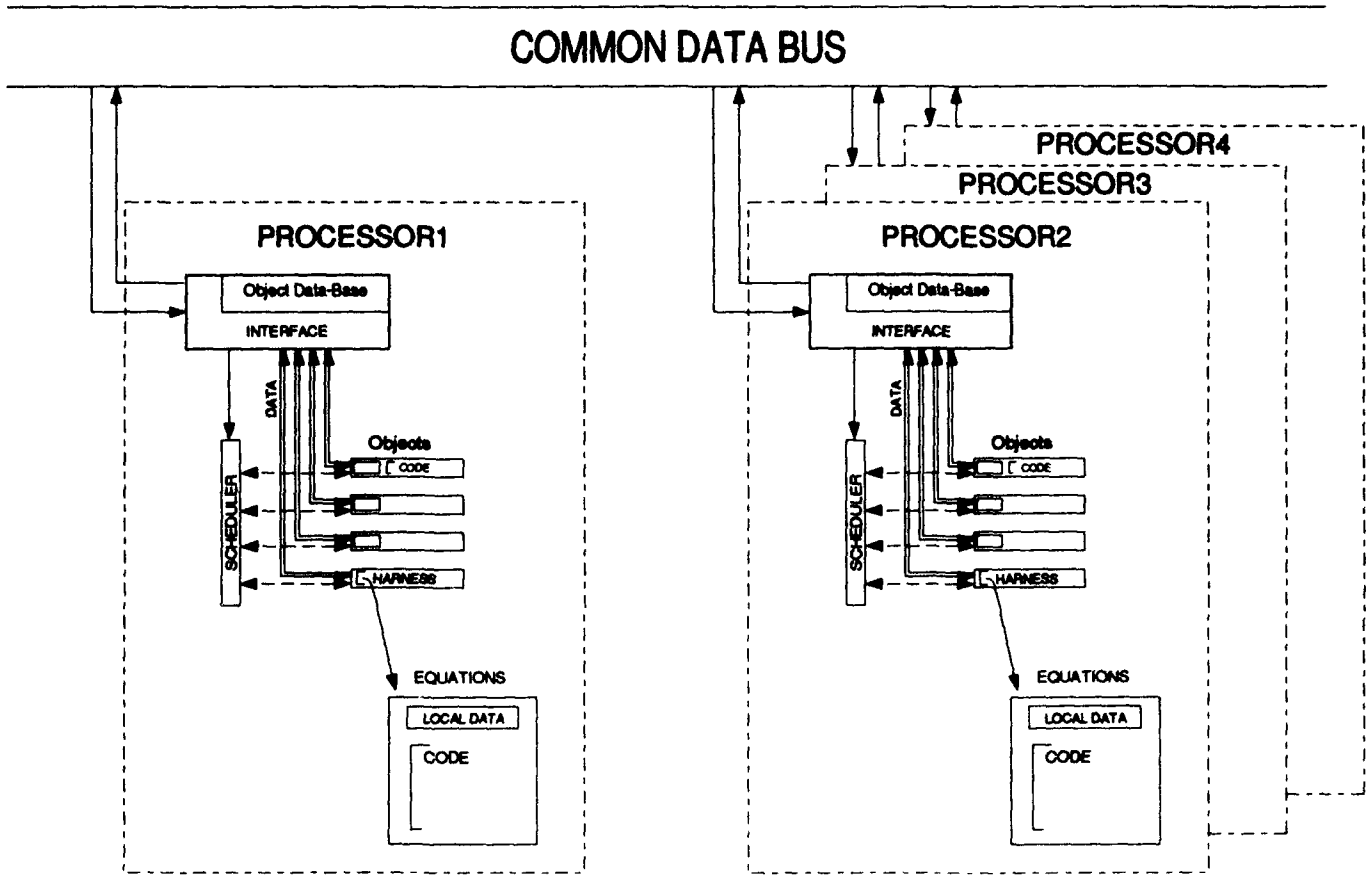


Fig 1. Software/hardware configuration for the object-based environment.

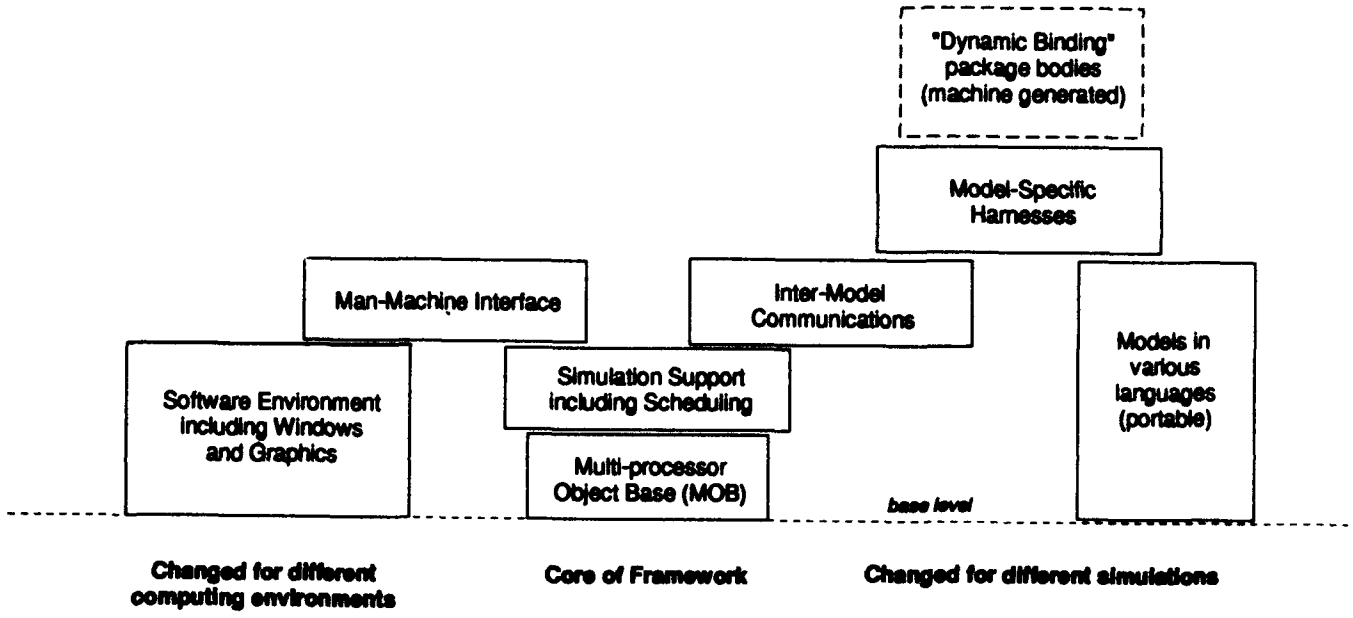


Figure 2. Major Software Inter-Dependencies within the Environment

DSSA-ADAGE: An Environment for Architecture-based Avionics Development

Louis H. Coglianese
DSSA-ADAGE Principal Investigator
IBM Federal Systems Company
Owego, New York 13827-1298
USA

Raymond Szymanski
E&V Project Manager
WL/AAAF-3, USAF Avionics Directorate
Wright-Patterson AFB, Dayton, Ohio
USA

1.0 SUMMARY

Advanced system architectures bring unprecedented capabilities to integrated avionics systems. To take advantage of the processors, system topologies, and algorithms, software architectures need to be open and flexible both to integrate new features into existing designs and to map applications onto new processing architectures. To date, development tools have focused on the means to make general improvements in productivity. Many good approaches in software reuse (e.g., CAMP), modeling and simulation (e.g., Matrix-X, Matlab) and CASE tools (e.g., RDD-100, Teamwork) concentrate on improving portions of the life cycle. The authors believe that, for avionics, it is necessary to extend and integrate these technologies: to move reuse into requirements and analysis, to smooth the transition from system and algorithm design and validation into real-time applications, and to use CASE tools' document generation and consistency management to flow design decisions into implementations.

This paper describes the Domain-Specific Software Architectures Avionics Development Application Generation Environment (DSSA-ADAGE) under development for the United States' Defense Advanced Research Projects Agency (DARPA) and the USAF's Wright Laboratory. It introduces the goals of the project, recent results in the development of a reusable software architecture for integrated avionics, a description of the process used to develop the architecture and an overview of the ADAGE development environment. The remainder of the paper is devoted to presenting the formal languages that describe the problem, solution and implementation views of the avionics architecture.

2.0 BACKGROUND

DARPA's Domain-Specific Software Architectures (DSSA) project is working to create an innovative approach for generating control systems. The goal is to use formal descriptions of software architectures, and advances in non-linear control and hierarchical control theory, to generate avionics, command and control, and vehicle management applications with an order of magnitude improvement in productivity and quality. Together with researchers from Massachusetts Institute of Technology, University of California at Irvine, University of Texas at Austin, and University of Oxford, IBM is developing an integrated environment for specifying, evaluating, and generating real-time integrated avionics applications. Focusing on Navigation, Guidance, and Flight Director, the project is defining an Avionics Knowledge Representation Language that specifies the features and constraints of avionics software architectures. The language will permit the non-procedural specification of applications and drive graphical representations of data and control. This approach relies on the ability to separate the architecture's problem-oriented features from its solution-oriented implementation constraints. It will allow a systems engineer to specify the system in domain-specific terms (filters, processors, sensors, rates) and let software composition and constraint-based reasoning tools provide the implementation details of scheduling and data access.

3.0 DOMAIN ANALYSIS PRELIMINARY RESULTS

An avionics system integrates the complex components of crew, airframe, power plants, sensors, and specialized subsytems into an intelligent airborne system for achieving specific mission objectives within time and space constraints. These specialized subsytems and their supporting avionics system capabilities require access to common time critical data produced throughout the airborne system with minimum, quantified delays to support complex subsystem dynamic stabilization, valid solution generation and valid fusion of varied data for eventual interpretation by crew members. Impediments to the availability of the time critical data are related to both the system architectural and system development requirements.

As new avionics architectures expand to include new complex subsystems, the processing required to meet hard real-time deadlines increases. New architectures are also responsible for creating wide variances in avionics computer hardware topology and associated data transfer requirements putting pressure on existing scheduling paradigms and communication mechanisms. In the current acquisition environment, and certainly in the future, physical characterizations of many system components are not available during the early stages of a development. Since all implementations of avionics systems are approximations of physical systems, developers are forced into an iterative development and refinement process of models and analyses which are often accomplished without automated tools to support the entire life-cycle.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

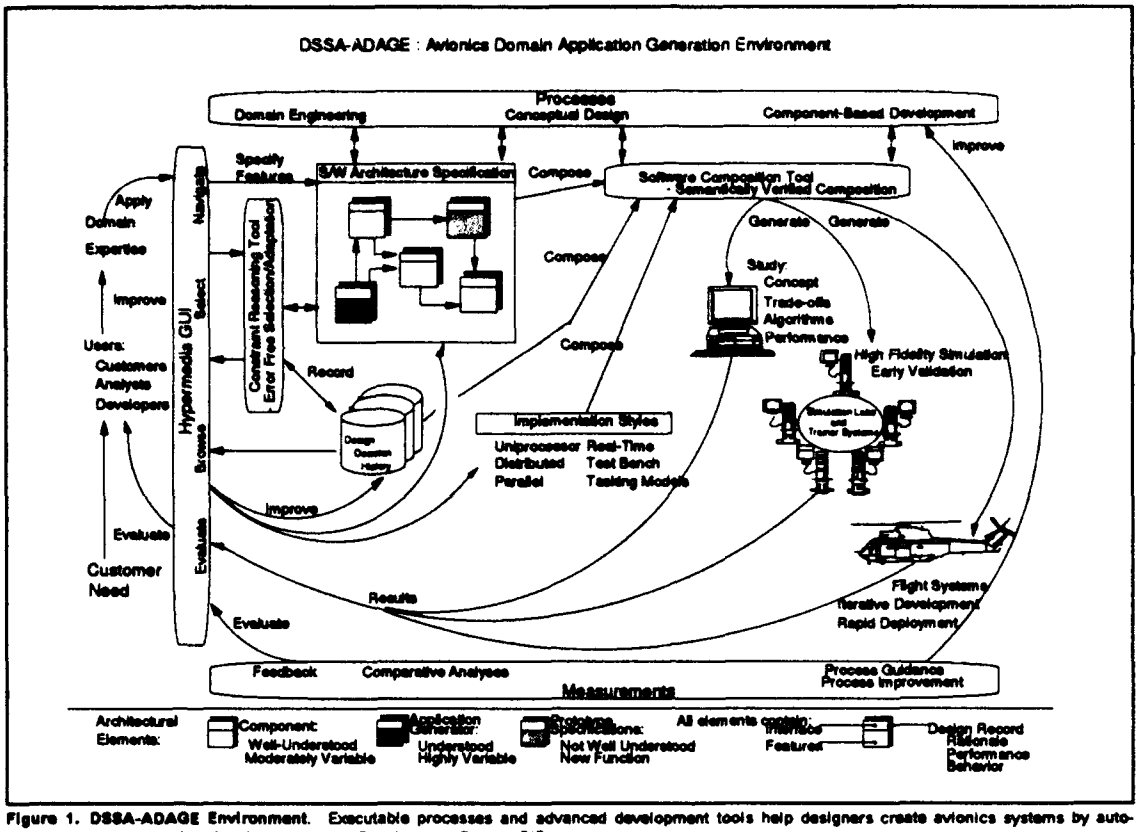


Figure 1. DSSA-ADAGE Environment. Executable processes and advanced development tools help designers create avionics systems by automating a spiral Architecture-based Development Process[1].

3.1 DSSA-ADAGE

The DSSA-ADAGE approach is based on the premise that many of the problems in Navigation, Guidance, and Flight Director are well understood. For any new system, several features will require new and innovative techniques but many components and subsystems can be built by combining and adapting existing solutions. Therefore, domain analysis can be used to identify components and constraints inherent in the avionics domain. Concepts in the domain can be organized both from the perspective of the physical problems that they solve and from the way the components work together in a computer program to solve them. This organization of components, connections, interfaces and behaviors is referred to as a Domain-Specific Software Architecture (DSSA). A DSSA not only provides a framework for reusable software components, but it also organizes design rationale and structures adaptability.

As part of DARPA's program the DSSA-ADAGE team is building an architecture and a hypermedia-based environment that assists analysts and software developers in automating avionics development. The ADAGE environment depicted in Figure 1 relies on:

- constraint based reasoning tools to reduce the user's adaptation and selection workload,
- software composition technology to construct analyses, models, simulations and real-time software for embedded systems by combining components and configuration data/parameters with implementation models, and
- a formal representation of iterative development process models and process measurement tools to guide user actions.

3.2 Domain Analysis Process

One of the primary goals of the DSSA-ADAGE project is to use domain analysis to engineer an architecture and associated components that can later be used to rapidly develop avionics requirements and software. This domain analysis is following a process that defines a mechanism for the orderly exploration of the problems and solutions in an application domain[2]. The process, shown in Figure 2 is based on several previously successful approaches to domain analysis[3, 4].

Process Step	Emphasis on
1. Bound the Domain	User's needs
2. Define Domain Specific Concepts	Problem space
3. Define Implementation Constraints	Solution space
4. Develop Domain Architecture	Module interfaces
5. Produce Reusable Workproducts	Implementation

Figure 2. Domain Analysis Process. The separation of problem-domain analysis from solution-domain analysis is a distinguishing feature of this process.

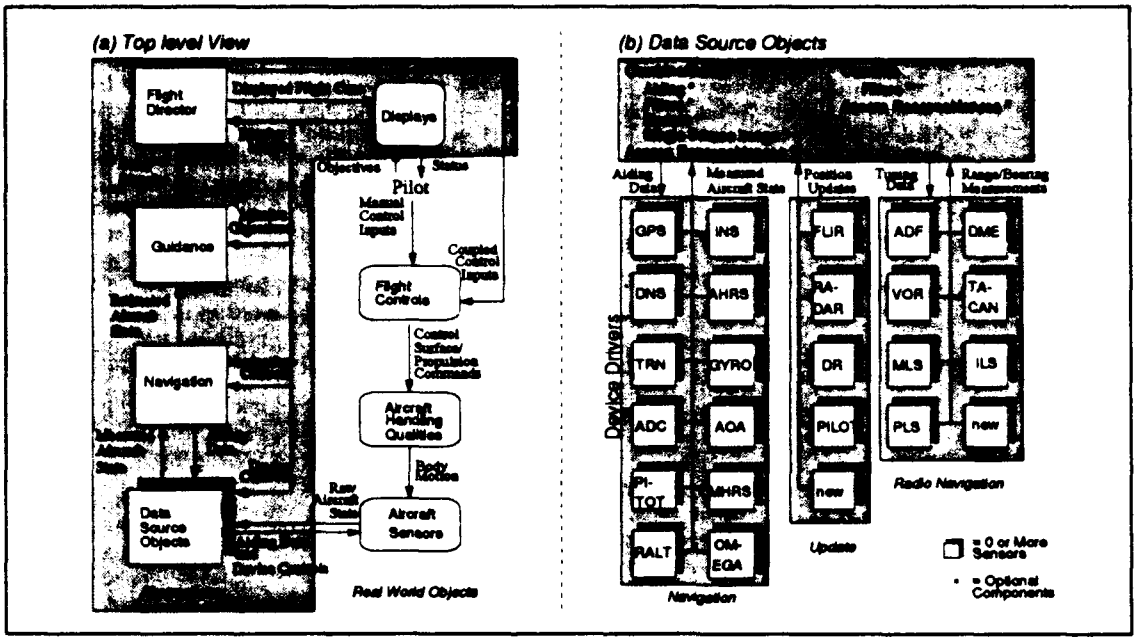


Figure 3. Pilot in the loop system. The top level view (a) of an integrated avionics system can be depicted as a series of layers that transform raw sensor data into control signals. A critical feature is the ability to combine data from a diversity of data sources (b).

The domain analysis process begins by bounding the domain and by setting goals for the analysis with the objective of defining an architecture and a set of components that cover a sufficiently large portion of the domain to significantly improve productivity and quality for avionics applications. The process then moves to defining the domain-specific concepts within the previously established bounds. This step's objective is to determine the central definition for the bounded portion of the domain. Once the features have been defined, the remaining challenge, before designing the architecture and the components themselves, is to define the implementation constraints. A primary activity during this phase is to quantify the range of configurability. At the highest level of configurability, the analyst needs to classify features as required, optional, or alternative. The final process steps, developing the domain architecture and producing reusable workproducts, are the production side of the process. They focus on defining interfaces, processing and configurability mechanisms that satisfy the requirements and constraints defined in earlier steps.

3.3 Domain Analysis Results

The domain analysis has resulted in the specification of high level Navigation, Guidance, and Flight Director architectures(see Figure 3.) The capabilities required for providing aircraft flight path management were assessed and allocated based on DSSA-ADAGE developed object-oriented definitions. An additional component, Data Source Object Driver, was identified as a necessary part of the Navigation domain. Appropriately, a Data Source Object Device Driver architecture has been defined.

The Navigation component determines aircraft position relative to one or more reference frames. This component's primary functions are to model the aircraft's operating environment and to integrate diverse sensor measurements into a single estimate. The current architecture permits the navigation component to adapt to a variety of data sources, filters, gains and earth and atmospheric models.

The Guidance component determines the difference between mission objectives and current aircraft state. It calculates a desired flight profile, estimates error in heading, speed and/or altitude, and assures smooth transitions between modes. The guidance architecture permits the guidance component to select the required modes, filters and gains, and to specify mode preconditions such as data quality, capture criteria, and mode conflicts.

The purpose of the Flight Director is to convert guidance errors into pilot control cues or autopilot commands. Its primary function is to develop cues based on errors, aircraft performance models and pilot models. As designed, the architecture can accommodate fixed or rotary winged aircraft parameters, varying aircraft flight models and pilot models, and different sets of control laws and gains.

The investigation of the aircraft navigation application domain has yielded systems with widely ranging system performance requirements, physical data sources, and real time processing requirements. Therefore, it was determined that a Navigation component reconfigurable design must incorporate a component that was capable of converting device specific data and protocols to

standard formats. This component, the Data Source Object Device Driver, acts as a buffer between the physical data sources and the navigation component. Its primary functions are to sequence through the legal states, monitor correct operation and control the physical device. This component is designed to select from a variety of functions, define device formats and protocols, select sampling rates, select filters and constants, and define quality criteria.

4.0 AVIONICS/ARCHITECTURE KNOWLEDGE REPRESENTATION LANGUAGE

> *Language is a reflection of a paradigm within a domain.*
>
> — John Goodenough[5]

Integrated avionics is an evolving domain challenged by the twin demands of expanding missions and advanced system architectures. The concepts used in avionics, however, are mature and well-understood leading to the conclusion that they can be further codified to support automated construction of avionics systems. While this notion is appealing, it ignores the problem that avionics knowledge encompasses a wealth of information from many disciplines. A design team coordinates knowledge of control theory, real-time scheduling, human factors, electrical design, and many other disciplines to translate customer needs into a working system. To coordinate this information a suitable language or coordinated set of domain-specific sublanguages must be used.

A sublanguage can be thought of as a way of expressing a user's point of view. Depending on the sublanguage, the statements can covey both formal and informal information about a system. A requirements cross-reference quickly asserts the satisfaction of customer needs. A system block diagram easily, but informally, identifies the system's hardware and software components and their inter-connections. To a control engineer, control block diagrams are a more formal, well-understood way of describing an algorithm. Finally, programming statements most formally express an algorithm at the expense of including large amounts of implementation details that often limit a reader's understanding. These different views of a system are, in a sense, complementary. While they include some of the same information, they also include different information appropriate to their levels and uses. Each is best understood by a different member of the design team. In the end, however, they must describe the same system.

The separate sublanguages, or views, need to exist so each discipline can express its aspects of the system in a familiar or comfortable notation. From a language point of view, the creation of a system from a domain-specific software architecture can be expressed by the equation:

$$System = DSSA \circ \sum_{i=1}^{n} Domain\_Statements_{Sub\_language_i}$$

ADAGE's domain analysis has brought out several conclusions regarding the types of knowledge that should be collected and how they should be used. ADAGE has focussed on the languages and tools that would assist an expert avionics engineer by eliminating many error-prone and mechanical steps in converting requirements into programs. ADAGE is designing formal languages for two purposes: to express the concepts embodied by the avionics-specific software architecture, and to form the basis for automation. Having formally specified the concepts allows avionics designers:

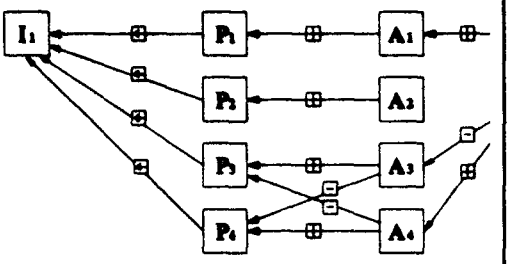
- to analyze the static aspects of the reference architecture,
- to see if the correct class of systems can be constructed,
- to determine if the right details are described, and
- to see if the descriptions easy to use.

This section outlines key elements of ADAGE's Avionics/Architecture Knowledge Representation Language (AKRL). Rather than giving a detailed description of all its features, it focuses on those areas that are important to the avionics domain. It is subdivided into sections that describe the organization of avionics knowledge into three views: the analyst's problem view, that develops strategies to meet customer needs; the architecture solution view, that converts the strategies into designs; and the architecture implementation view, that implements the design. One or more sublanguages are required to describe each of these aspects of an avionics system. These sublanguages are designed to have both textural and graphic representations that speak in terms appropriate to their users.

4.1 Analyst's Problem View

The highest level view of a system, the description of the strategies used to satisfy requirements, contains the broadest and deepest knowledge. While it is clearly infeasible with the state of the art to automatically design an avionics system by simply evaluating customer needs, it is appropriate to capture strategies used by expert designers. Since an avionics software architecture embodies a class of avionics solutions, the way each system uses it to satisfy its requirements may vary. ADAGE needs a way to record how designers have used the architecture to meet typical requirements. The information includes a customer's need (an issue), a description of one or more alternate ways to solve it (a set of positions) and rationale about when each strategy should be used or when it another is preferable (arguments).

ADAGE is using the Issue Based Information System (IBIS)[6] notation to record the problem view of the avionics system(see Figure 4). IBIS describes a network of information that builds evidence for taking positions on issues. This information can be used to provide general guidance to the avionics designer for considering alternative designs. It also can provide a checklist of typical solutions that define an organization's product




```
ISSUE_1:    Required position sensors
POSITION_1: 4 Channel GPS
ARGUMENT_1: Highly Accurate and does not require
POSITION_2: 1 Channel GPS and INS
ARGUMENT_2: Highly Accurate but requires another
POSITION_3: TRN and Laser Altimeter
POSITION_4: TRN and Radar Altimeter
ARGUMENT_3: Good accuracy required
ARGUMENT_4: Covert operation required
```

Figure 4. Subset IBIS description of sensor selection. *gIBIS graphical notation allows users to follow arguments and positions concerning design issues. The textual representation can be readily updated or included in the design rationale.*

line. Users are not limited to accepting the dictates of past systems. When new requirements or new algorithms become available, designers are free to add new issues, positions, or supporting arguments to the knowledge base.

In the ADAGE environment the designers' decisions and their rationale are recorded automatically. This high-level rationale is reported as documentation for peer reviews and for long-term maintenance.

4.2 Architecture Solution View

Architectures have often been depicted by layer diagrams, data-flow diagrams or block diagrams. Proponents of Object-oriented Analysis[7, 8] (OOA) have developed means of describing the *classes of objects* in a domain and their operations, *associations* between them and the *constraints* upon them. Their notations create a model of the domain data that represents a portion of a expert's knowledge. ADAGE's domain analysis used a combination of two object-oriented analysis techniques[4, 9]. It identified classes, associations and constraints for the integrated avionics domain. Rather than using an existing notation, the objects in its avionics architecture have been represented by equivalent notations that can be understood by the ADAGE environment.

The first part of the architecture represents the classes of avionics objects, the constraints on their number and their data type dependencies. ADAGE uses a set of *parameterized type expressions*[1] , to define a layered view of the architecture. Each layer in the system is defined as a *realm* of plug-compatible components. All members of the realm are required to output data of the same type although they may input data of different types. ADAGE's portion of the avionics domain is concerned with estimating aircraft state based on a suite of sensors, a set of mission objectives, a collection of filtering algorithms and their relationships. The system has been represented as a set of layers, from data sources at the bottom, through navigation, to guidance and flight director at the top. Each layer, transforms its data into a form usable by the next higher layer. For example, each sensor reports the raw data in its own coordinate frame. The data source layer converts the raw data to a standard aircraft state estimate in a common coordinate frame. The navigation layer refines the estimates into a system estimate with respect to the coordinate frames needed by guidance and by the crew.

In the case of navigation, the realm defines the stages of sensor data combination and filtering that can be integrated into the system. The example below shows a simplified subset of the navigation realm.

```
NAV  = {derived_data[i:INAV],...}
INAV = {compfilter[i:INAV,d:DPLR],...
        auto_selection[i:{INAV}],
        gps_ins[g:GPS,i:INS],...
        dplr_ins[d:DPLR,i:INS],...,
        gps[g:GPS], ins[i:INS],...}
```

It states that NAV contains, among other things, an algorithm for deriving earth-referenced aircraft state parameterized by the realm of internal navigation data INAV. It also states that the data type output by INAV can be created by an INS, by a GPS, or by some combination of the two. The ellipses indicate this example only defines a subset of the components in these realms. An interesting feature of type expressions is that components in a realm may be *reflexive*, i.e., they take as input other components in the realm. In the example above, the compfilter can take any INAV output and combine it with a DPLR before outputting another INAV output. As an example, the expression in Figure 5 describes a simple system in which the reflexive component auto_selection chooses among three INAV components, at run-time, before passing the result to derived_data.

Object classes and layering type dependencies are not sufficient to describe the architecture at this level. The type expressions lack a means for describing the data flow between components (functional model in OOA terms) including temporal (throughput and consistency) requirements between components and the quality of the data on the interfaces. The *software circuit* paradigm[12], developed by David McAllester of the Massachusetts Institute of Technology, defines constraints between components in an analogy based on clock and data wires in electronic circuits. Using the circuit definition, the ADAGE environment can perform

[1] The concepts of *parameterized type expression*, *realm* and *reflexive component* were developed by Don Batory from the University of Texas at Austin on the Genesis project[10, 11].

(a) Textual view

```
Simple_NAV =
   [derived_data
     [comp_filter
       [auto_selection
         [gps_ins[Some_GPS,Some_INS],
          gps[Some_GPS],
          ins[Some_INS]]]],
      dns[Some_DNS]
```

(b) Hierarchical view

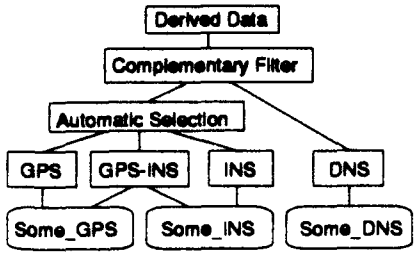


Figure 5. Type Expressions. ADAGE graphical user interface tools convert the textual notation into a more readable form. While the the ADAGE tools use the textual notation (a) to describe the layering of the architecture, users prefer data flow diagrams, hierarchy charts (b) and menus.

several analyses. First, it can check that the circuit obeys design completeness and consistency rules. It can construct models of the noise present in the estimates of real world parameters such as the aircraft estimated state. Finally, using models of execution times, it can verify real-time performance requirements. These constraints were singled out because the domain analysis indicated that they would provide the greatest benefit to the developers. For example, the combination of performance and noise modeling will permit ADAGE to suggest certain components would be more suitable than others for a given design. Performance modeling coupled with lower level scheduling paradigms (e.g., Rate Monotonic, Earliest Deadline, Cyclic) would spot timing problems before the system left the designer's desk.

There are many other constraints that exist at the analyst's level view of an architecture. Even a simple constraint such as "Terrain Following requires a forward-looking altitude data source with a range of [aircraft_specific] miles and and accuracy of [x]" defines functional dependency between components, constraints on the attributes of components and dependencies between components and system models. ADAGE uses Ontic[13], a language for describing mathematical concepts, system specifications, implementations, and verifications, both to implement the *circuit compiler* and to represent first-order logic constraints on architectural elements. One of Ontic's primary advantages is that statements written in the language can be evaluated using a non-deterministic version of LISP. Therefore, the ADAGE's constraint-based reasoning system can evaluate Ontic descriptions of constraints to assist the user, via the ADAGE graphical user interface, in selecting models, components and numerical parameters. Since the syntax of the language is close to LISP and since the expressions deal with low-level details of the architecture, the application developer will not interact directly with the Ontic representation.

4.3 Architecture Implementation View

Perhaps the most important concept in object-oriented analysis is the definition of the class inheritance hierarchy. ADAGE uses LILEANNA[14], developed by Will Tracz of IBM, to specify class hierarchies and to compose them into Ada packages. LILEANNA – LIL Extended with ANNA (Annotated Ada) – is a module composition language for designing, structuring, composing, and generating software systems in Ada. LILEANNA extends Ada by introducing two entities: *theories* and *views*, and by enhancing a third, package specifications. A LILEANNA package, with semantics specified either formally or informally, represents a template for actual Ada package specifications. It is used as the common parent for families of implementations and for version control. A *theory* is a higher-level abstraction (a concept), that describes a module's syntactical and semantic interface. A *view* is a mapping between types, operations, and exceptions.

Programs can be structured and composed using two types of hierarchies: *vertical* (levels of abstraction and stratification) and *horizontal* (aggregation and inheritance). LILEANNA supports this with two language mechanisms: import dependencies called *needs* and three forms of inheritance called *import*, *protect*, and *extend*.

Figure 6 demonstrates how ADAGE uses some of these features to represent avionics concepts. Integrated avionics systems often require a data selection mechanism based on the quality of data from the input sources. There are, however, many ways of describing the quality of information coming from a data source. Rather than coercing users into choosing one representation, ADAGE defines a theory of data quality. This permits the architecture to define elements that depend on the existence of the concept of data quality without burdening them with details of any one implementation. In the class hierarchy the theories of data quality and the aircraft state vector are merged to create the concept of measured state i.e., an estimate of the aircraft's position, velocity and attitude as measured and qualified by a data source. The diagram shows just one use of measured state, its input to selection routines to choose the appropriate source for a particular element of the state vector. To create an automatic selection routine, a user would chose one of the selection routines, e.g., automatic regression, and one model of data quality. The software composition tools in ADAGE would collapse the class hierarchies to produce an optimized regression routine.

There is a clear mapping from LILEANNA's formalisms to the *parameterized type expressions* dis-

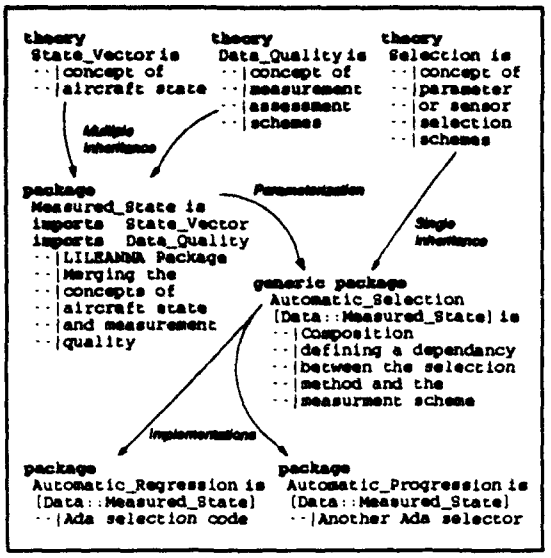


Figure 6. Sample LILEANNA Relationships. Theories and LILEANNA packages provide the abstraction mechanisms to compose type expressions into Ada packages.

cussed earlier. LILEANNA packages and *theories* correspond to *realms*. with the exception that, in LILEANNA, the software composition mechanisms (inheritance and parameterization) are explicit.

LILEANNA contains a second feature not found in *parameterized type expressions* - ANNA[15, 16]. ANNA lets users define behavior in terms of first-order predicate logic. Behavior *statements can specify invariants* and constraints on any level of object from a *theory* down to a Ada package. The behavior can be validated at run-time because the ANNA tools translate the assertions into pre-condition and post-condition checking code. Since all constraints cannot be verified at design time, The ADAGE environment intends to use ANNA to support algorithm validation during desktop simulations. The run-time checks can be eliminated for the real-time embedded system.

5.0 SUMMARY

Domain-specific software architectures can provide the structure for the improved automation of integrated avionics systems development. They can facilitate *megaprogramming*[17] — the process of constructing software one component at a time, rather than one line at a time The avionics domain exhibits the attributes [18] necessary to demonstrate the viability of this approach for real-time system development, maintenance, and evolution. Two challenges remain. First, avionics needs well-engineered components that can be readily adapted for use in a variety of airborne systems. In addition, to fully meet the challenge, there needs to be an open environment (ADAGE), based on domain-specific languages, that organizes these components within the context of a software architecture and provides analysis and synthesis tools to facilitate the megaprogramming process.

6.0 REFERENCES

1. Coglianese, L. and Tracz, W., Architecture-Based Development Process, Owego, NY: IBM Federal Systems Company, ADAGE-IBM-92-03, 1992.

2. Tracz, W. and Coglianese, L., Domain-Specific Software Architecture Engineering Process Guidelines, Owego, NY: IBM Federal Systems Company, ADAGE-IBM-92-02, 1992.

3. Prietro-Diaz, R., Reuse Library Process Model, Software Technology for Adaptable Reliable Systems, AD-B157091, IBM CDRL 03041-002, July 1991.

4. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Pittsburgh, PA: Software Engineering Institute, CMU/SEI-90-TR-21, November 1990.

5. Carlson, W. and Vestal, S., Editors, Proceedings of the Joint Domain-Specific Software Architectures and Prototyping Technology Workshop, Steamboat Springs, Colorado: Defense Advanced Research Projects Agency, January 1993.

6. Lubars, M., Representing Design Dependencies in the Issue-Based Information System Style, Austin, Texas: Microelectronics and Computer Technology Corporation, STP-426-89, November 1989.

7. Rumbaugh J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design* Prentice Hall, 1991.

8. Booch, G P., "Object Oriented Development," *IEEE Transactions on Software Engineering*, March 1986.

9. Coglianese, L., McIntyre, H., DSSA-ADAGE Domain Analysis using OMTool, Owego, NY: IBM Federal Systems Company, ADAGE-IBM-92-12, 1992.

10. Batory, D.S., Building Blocks of Datbase Management Systems, Austin, Texas,: University of Texas, TR-87-23, February 1988.

11. Batory, D.S., Construction of File Management Systems from Software Components, Austin, Texas,: University of Texas, TR-87-36 REV, October 1988.

12. McAllester, D., An Adage System Vision, Cambridge, Massachusetts: Massachusetts Institute of Technology, ADAGE-MIT-92-02, 1992.

13. Givan, R., McAllester, and D., Zalondek, K., ONTIC91: Language Specification User's Manual, Cambridge, Massachusetts: Massachusetts Institute of Technology, 1991. Draft 3.

14. Tracz, W., "LILEANNA: A Parameterized Programming Language," *Proceedings of Second International Workshop on Software Reuse*, March 1993.

15. Luckham, D.C. and von Henke, F.W., "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, pp. 9-23, March 1985.

16. Luckham, D.C., von Henke, F.W., Kreig – Brueckner B., and Owe O., *Lecture Notes in Computer Science, no. 260: Anna, A Language For Annotating Ada Programs, Language Reference Manual* Springer-Verlag, 1987.

17. Boehm, B., "DARPA Software Strategic Plan," *Proceedings of ISTO Software Technology Community Meeting*, June 27-29 1990.

18. Mettala, E.G., Domain Specific Software Architectures presentation at ISTO Software Technology Community Meeting, 1990.

ENTREPRISE II:

A PCTE INTEGRATED PROJECT SUPPORT ENVIRONMENT

Gérard OLIVIER
EII Software
315 bureaux de la Colline
92315 SAINT-CLOUD CEDEX
FRANCE

1 - ENTREPRISE II : THE DEVELOPMENT ENVIRONMENT FOR MAJOR SOFTWARE

Entreprise^{TM} II users are in the technical and scientific software industry and include members of software development and maintenance teams, at all levels: administrators, project managers, project leaders, those in charge of sub-projects or tasks, developers and maintenance staff. Entreprise II is also designed for software tool editors, who need an integrated CASE tools environment in which they can develop and distribute their own tools.

Developers of software systems face numerous problems directly related to software development, upkeep and maintenance. Entreprise II provides a solution to these problems, at both organizational and technical levels.

1.1 - Controlling the organizational factors

This involves defining, formalizing, implementing and monitoring the development of software applications development to be integrated in a system whose installation requires complex structures and the cooperation of specialists from different fields (concurrent engineering).

These activities take place in an industrial environment which is heavily influenced by the following:

- world-wide structure of large organization,
- need for international cooperation between organizations,
- geographical distribution of development sites,
- difficulty of managing complexity, organization, cost, production and installation deadlines of the projects,
- diversity of projects and capabilities involved in software production,
- diversity of projects undertaken by any organization.

1.2 - Controlling technical factors

It is important to control the factors affecting software development quality in the following areas :

- increasing complexity of software applications,
- rapid technological improvement,
- increasing diversification of software applications, which results in ever greater demand and consequently the need to increase the developers' individual and collective productivity,
- extended life of software applications, resulting in the need to prolong the period of software maintenance and to increase investment in maintenance,
- increasingly strict quality requirements, due to the introduction of software in the critical parts of sensitive or high-risk systems.

1.3 - Controlling the maintenance factors

The increase in maintenance activities is forcing software developers to find ways to automate and support the maintenance tasks. The reluctance of software developers to devote time to these tasks (they generally prefer to focus on development), as well as the lack of qualified software developers to meet development demands, contribute to the growing imbalance between development and maintenance. Large organizations have to look for solutions which improve automation of development and maintenance. Method tools, formalization tools and support tools for software development and maintenance techniques existing today on the market are extremely diverse.

Therefore, the first problem to tackle in the process of automation is this diversity of tools, which can be overcome by modelling and formalizing the organization of the development and maintenance processes.

Presented at an AGARD Meeting on 'Aerospace Software Engineering for Advanced Systems Architectures', May 1993.

1.4- CASE tools fragmentation

The current range of Case tools is characterized by:

- the existence of varying and incompatible basic methodologies,
- numerous differing, incomplete and unrelated methods,
- a disorganized range of tool-type products.

This fragmentation in methods and tools appears to be the most serious factor hindering from production and quality control. The level of investment required to obtain a total solution may be one reason for this fragmentation. Each tool supplier tends to limit itself to solving problems relating to a single part of the development process. Thus we find a large number of specification, design, programming and test tools, but a total lack of complete environments.

This situation, in turn, results in fragmented software development practices and often forces developers to perform "manual" transitions between phases in the software life cycle. For instance, the lack of automation in the transition between the design and coding phases means that developers have to provide documents proving the progression of the design phase and the justification for moving to the coding phase. These documents form "links" and "reference points" between phases, which should be available for re-use in maintenance phases or when backtracking from coding to design. These manual procedures generally make the development process inefficient and also complicate the maintenance process.

It is essential to find solutions which enable integration and automation of the entire development process.

1.5- Horizontal and vertical activities

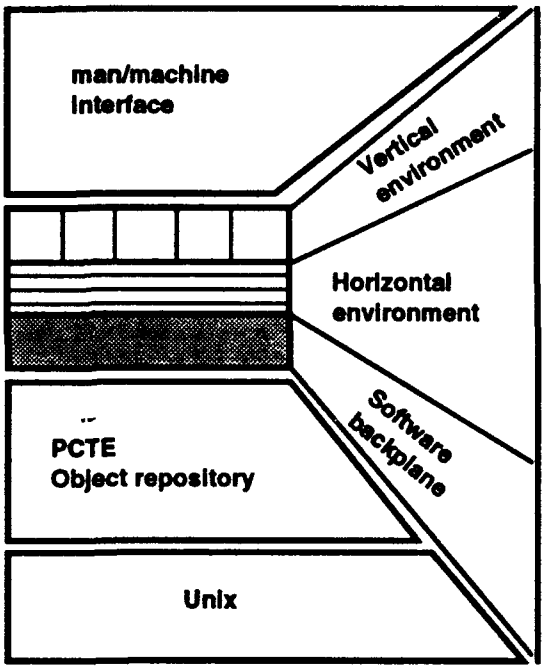
The software development process includes two types of activity:

- life-cycle activities (simulation, prototyping, specification, design, coding and testing).
- cross-life cycle activities activities performed during development (project management, configuration management, documentation, quality, re-use, etc.).

In order to integrate the development process we have to define the general frame for these activities and in particular decide which data will be used and by which methods the data will be handled in terms of rules, rights and responsibilities. The automation of the development process calls for the creation of tools which can support both horizontal and vertical activities and transitions between activities.

The facilities and architecture of Entreprise II have been designed to meet these support requirements for the whole range of development and maintenance activities for large software applications. The design stage involved collaboration with a number of major technical French users (Thomson CSF, Aérospatiale, Dassault Aviation, Dassault Electronique, Sextant Avionique, Sagem, Sfim) and the Délégation Générale pour l'Armement.

2- ENTREPRISE II, A LAYERED ARCHITECTURE



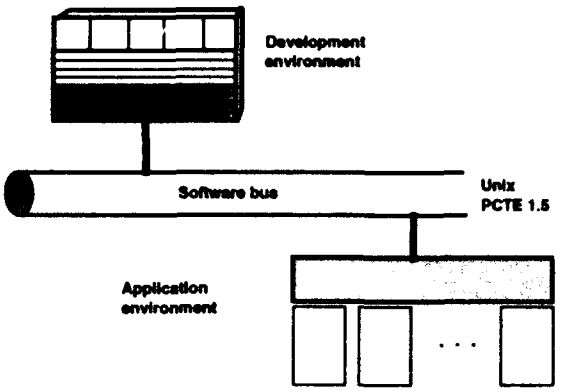
A open layered architecture

Entreprise II is based on a layered architecture. The first layer includes the software framework, providing interface facilities with the various tools in the environment and complying with a data communication protocol. The next layer contains facilities for "integrate" software tools. These facilities can be grouped together under the generic term "backplane". The whole system used to integrate CASE tools forms the Integrated Project Support Environment (IPSE). When vertical software engineering tools of various origins are inserted into this open environment, it becomes a Populated Integrated Project Support Environment (PIPSE). The system always uses the same look and feel.

2.1- PCTE: The Entreprise II software framework

The software framework is the communications channel for all the Entreprise II components. It uses the PCTE 1.5 (Portable Common Tool Environment) standard. Entreprise II will be ported to ECMA/PCTE.

The software framework is a critical element for the interaction of software systems, whether they are development environments or applications developed and maintained using these environments. It is essential that the software framework should act as a standard, guaranteeing long system life and attracting suppliers of tools, environments and applications.



The software framework: the key to system interaction

2.2- The software backplane

The software framework alone, whilst essential for an integrated environment, is not the only prerequisite. The homogeneity of the environment's methods and tools requires compliance with a set of principles and rules implemented by the Entreprise II backplane.

This backplane is supported by PCTE and makes it possible to formalize the development-maintenance process :

- integration of the organization in the process,
- setting up effective communication between all teams and organizations, particularly to accelerate the decision-making process,
- setting up the means to follow up and inspect the development steps,
- adaptation of the methods and tools used to the specific requirements of the organization, projects and individuals,
- need to integrate a coherent quality control policy which aims to increase the automation of inspections.

All these requirements must be solved within a coherent frame for which foundations must be laid. This is the role of the Entreprise II backplane. Without this backplane methods or tools assembled within an environment would be heterogeneous and incoherent.

PCTE itself defines a number of standard and public data schemas. Entreprise II thereby contructs a philosophy for handling development and maintenance. The Entreprise II schemas define three types of data dictionaries:

- Nomenclature type dictionary :covering all data for managing the IPSE and information concerning methods ;
- Encyclopedia : for each project developed in the IPSE which contains all data produced during development and maintenance;
- Reusable Objects Data Dictionary : containing all data which can be re-used from one project to the next.

2.3- Managing the dialog with the user

Entreprise II is an interactive development environment. The dialog is standardized whatever tools are employed by the user.

When the user connects to the workstation, he enters a session, i.e. an environment composed of an IPSE, a project, a task and a role:

- at the project level, Entreprise II defines the tasks on which the user can work,
- at the task level, Entreprise II determines whether the user belongs to the group of users allowed to participate,
- at the role level, Entreprise II determines the tools to which the user has access rights and allows him to customize the presentation of these tools.

These measures contribute to the security of environment functions according to the following basic principle:

the only functions presented to a user are those for which he is authorized.

One user may be connected to various tasks belonging to various projects within a single environment. The user can only activate the tools associated with the task on which he is working.
The command language and the graphic navigator are used to navigate around the data dictionaries and activate the tools (with a display or graphic table for selection of options and start-up parameters). Graphics are used for dialog with the user. Expert users may alternatively use a

text command language for dialog. The look and feel of the Entreprise II dialog complies with the Motif™ or Open Look™ standards. The open-ended design of Entreprise II's man/machine interface means that it can be adapted to other look and feel standards.

2.4- Structuring the development

Entreprise II enables its users to base their development structure on four simple, yet powerful concepts:

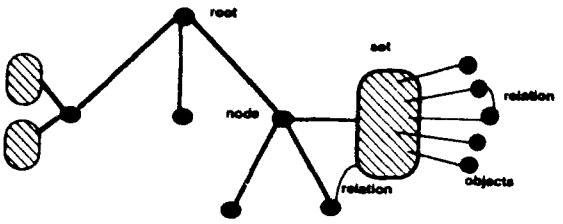
- the tree (root + nodes) built by the user, which forms the general frame. The participants will look here for the information required and will place the objects they produce in this frame,
- the object sets, positioned on the tree nodes,
- the objects, arranged in sets,
- the relationships. The user or the tools used may establish relations between the three types of entities. This will be done in accordance with the semantics defined and in compliance with the basic rules imposed by Entreprise II.

These trees, sets, objects and relationships can also be used to define the principles for navigating within the project data base (encyclopedia). The user can add attributes to these entities to facilitate navigation and selection.

This tree-structure is used for project management, document management, configuration management, re-usable objects, specification, design, code, etc. The "tree" concept has enabled the creation of a unique graphic display and navigation system. The system for all the functions of Entreprise II, whether original or added, is independent of the tools and very efficient. The project methodologies are also based on this model. In fact, methods such as DoD 2167A, DO178B or GAM T17 V2 actually impose a tree-structure organization of developments.

As an example, Entreprise II offers software developers the possibility of building their own structure to conform to the breakdown of the software they have to produce. In this way, the reference tree used by the project reflects the software developed:

- the root of the tree corresponds to the software,
- the breakdown of the software into software elements (and subsequent breakdown of these software elements into other elements or components), results in the development of the complete tree,
- the software objects produced (e.g. specification and design documents, diagrams, code, binary data etc.) are organized into sets and positioned on this tree.



Four concepts for a general structure

This structure, defined by the user, is the working basis for all the tools integrated in Entreprise II. Tools must therefore position within this structure all data that can be shared with other tools. This data can be of any type (text, graphics, images, etc.).

2.5- Controlling data access

The nomenclature database is used to carry out a number of checks concerning data confidentiality. These checks depend on:

- the data schemas, loaded at connection, which define the user's view of the project data dictionaries.
- the tools available to the user during a session,
- the access rights used by the tools to control user action,
- the user's access rights.

This security also helps to increase productivity, as it eliminates many types of potential errors.

2.6- Managing the versions

The encyclopedia database associated with each project contains all the objects produced during the project (documentation, sources, binary data, technical notes, etc.). A standard data set is associated with each object in the encyclopedia. Each tool or user can customize its/his view of the objects by defining and adding the data and relations it/he requires. The following basic functions are available to the user for handling trees, sets, and objects:

- *editing*, for modifying an entity,
- *stabilization*, for prohibiting modification of selected entities,
- *duplication*, which allows several users to work simultaneously on the same entities.
- *synchronization/delivery*, which keeps the user informed of modifications performed on these duplicated entities by other users.

These functions provide coherent version management in all development activities, regardless of the type of objects handled by users.

2.7- Administering the IPSE

Entreprise II is used for administration of the environment, managing security and confidentiality of *all the components (methods, tools, participants,* products, etc.).

The backplane is the host structure for CASE tools used in the environment. It defines the general frame for the development and maintenance processes, including:

- method : before using Entreprise II, the project leader defines the methodological frame for development, which allows the environment to be configured accordingly. Entreprise II can be used for all standard methods and also allows the project leader to define his own method. The incorporated default methods are GAM-T17 (V2), MIL-STD DoD 2167A and DO 178B.
- organization of the software life cycle,
- company and team practice,
- tools used in the different phases of the software life cycle,
- man/machine interface: Motif or Open Look' based on the X.WINDOW™ standard.

3- FEATURES OF ENTREPRISE II IPSE

The Entreprise II integrator development environment supports the facilities which are common to all phases of the software life cycle. The three main productivity factors in major software projects are configuration management, documentation management and project *management.*

3.1- The Configuration Manager

Configuration management means management of product versions at the development stage (changes), or maintenance stage (evolutions). The Configuration Manager is used to automate the following:

- configuration of all or part of the software (definition of a version),
- generation of a version,
- consultation of a version,
- management and archiving of versions (history, dependency, etc.).

The following four types of standard entity are provided for managing modifications:

- software problems reports sent by end-users of a version. Incorporation of modifications to response to these reports can be refused, delayed, or accepted.
- evolution requests intended for the maintenance teams. These requests represent requests for modification that have been accepted.
- change requests attached to a version, which include all modification sheets incorporated during the development of a version.
- modification sheets.

The Configuration Manager can be used for all industrial methods and practices. *It allows the user to trigger* operations (defined by the user) when the status of these entities changes. It also allows the user to define other types of entities and other statuses or types of transition between statuses.

These customization possibilities, which are vital for incorporating the specific characteristics of each company or project, makes the configuration management tool the prized partner of software maintenance teams.

3.2- The Traceability tool

The Tracability tool allows software modules to be followed throughout the different phases of their life cycle. The Tracker provides a horizontal view of software production, through the various tools used for specification, design, coding, and tests. It is particularly valuable for tracking software specifications throughout the different stages of development, i.e. for recognizing to which part of the specifications a development object (e.g. design diagram or code part) is related.

3.3- The Documentation Manager

The Documentation Manager automates production and management of the technical documentation relating to a project. Where a certain development method has been associated with a project, a documentation diagram allows the documentation to be automatically structured according to the same method, using the software breakdown tree.

The Documentation Manager automates document production and ensures its coherence, using standard DTP (Desktop publishing) tools. The Documentation Manager tool currently uses TPS™ and/or Framemaker™ tools, as selected by the user. It is an

open system, allowing other tools to be integrated, as required by the user.

3.4- The Project manager

The Project Manager tool provides help with all project supervision tasks, using environment data:

- structuring,
- organization,
- estimation,
- planning,
- scheduling,
- follow-up.

The Project Manager is interfaced with the ARTEMIS™ product which may be used as a project management tool for the system featuring the software developed or maintained using Entreprise II.

The integration of ARTEMIS in Entreprise II enables data to be exchanged between the two environments and checked for consistency.

3.5- Code production

Coding activity depends not only on the programming methods but above all on the programming language and the production technology used (editors, compilers, linkers, loaders, symbolic language debugger, library manager, etc). With the Entreprise II production tool, coding activities do not depend on the production technology used, as its programming environment provides coherence between the different tools.

Entreprise II currently integrates various production (compilation) technologies for ADA, C, C++ and LTR 3 languages. It provides the programmer with a source organization model which is independent of the programming language used. It gives programmers the means to navigate and graphically display the source texts of code objects. It stores in memory the relationships existing between source objects (at input) and/or binary data (after compilation) for one application. It ensures, if required, consistency between all application objects.

For all programming languages supported, it provides compilation or automatic re-compilation of the application. It automatically creates executable applications. It provides the programmer with a powerful multi-language syntax editor for inputting his source objects free of syntax errors. It manages interdependency of source objects (ADA, C++, C, LTR3, etc.) so that transformations (editions, compilations, etc.) can be applied. These language-specific transformations are defined when a compilation technology is connected to the production tool. It manages the program production environment, e.g. checking the presence of library interfaces for the production of ADA binary data, or that files returned to a C source by *#include* commands are present.

The Entreprise II production tool is an open system, allowing various compilation technologies to be integrated. The list of technologies integrated so far is not exhaustive - other technologies will be integrated to meet the needs of users.

3.6- The Reusable Object Dictionary

Sometimes some objects (source programs, document frames) developed during one project need be reused for another project and stored in a specific data dictionary. The Reusable Object Dictionary manages objects common to all projects in the environment. Its data are stored using a theme tree. Facilities available are:

- archiving, which allows the recording of objects considered to be re-usable,
- consultation of the dictionary, with a search function for selecting re-usable objects,
- extraction, which allows the re-usable objects selected to be inserted in the current project encyclopedia.

This tool forms the basis for software re-use operations.

3.7- The Communication Manager

Three levels of communication are provided by Entreprise II:

- E-mail between users (software development or maintenance teams) working on the same project or on different projects in the same IPSE,
- composition, broadcasting to lists of users and display of notes, agendas, minutes of meetings, etc...
- communication of objects between project databases located in remote installations.

This set of tools provides the developer with a real office system environment.

4- FEATURES OF THE PIPSE

Entreprise II covers all phases of a software product's life. It is an environment open to all types of vertical tools and standardizes currently available horizontal tools. There are two methods of integrating CASE tools into the backplane :

- loose integration,
- tight integration.

The backplane has a standard "toolkit" for loose and tight integration of CASE tools. This allows users to integrate the tools they wish to use in Entreprise II, and enables the horizontal services to be applied to the objects produced using these tools.

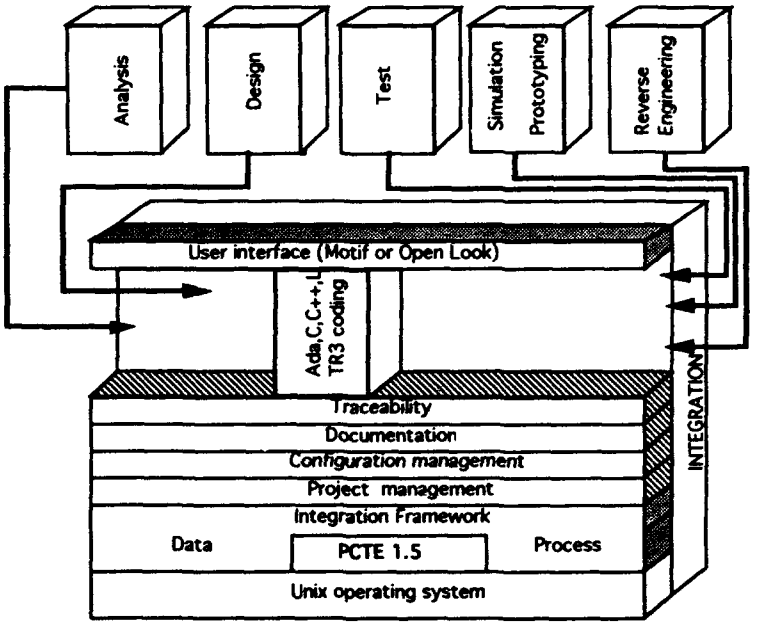
The operating dialog respects either the Open-Look or Motif standard, as required by the user. Entreprise II provides tool integrators with a module to help them design and create these dialogs.
Each tool uses in-built access rights to carry out checks enabling it to adapt its operation to the user's requests and access rights. Entreprise II provides services for developing interactive tool dialogs. Dialogs can thus be created on X.Window which respect the selected "look and feel" standard.

The Entreprise II backplane thus offers a number of services for developing integrated vertical tools, *including:*

- a specific user environment
- a dialog manager
- a window manager
- services providing access to data dictionaries,
- all PCTE facilities,
- an integrated command language (Shell),
- a graphical navigator

Buying a new development and maintenance package will not mean that previous investments will be wasted, as Entreprise II can host existing tools, making them usable in an Entreprise II workbench without implantation on PCTE. For this, Entreprise II can be used to create host capsules (loose integration). Entreprise II allows data exchange between a project encyclopedia and the host operating system (UNIX™). For example, host system files can be retrieved in an encyclopaedia, and encyclopaedia objects can be communicated to the host system. These exchange services are particularly useful for hosting tools.

Different software manufacturers use different vertical tools and related methods. To allow for this, Entreprise II can accommodate all vertical tools required by its users, whether these tools are commercially available or owned by the manufacturers themselves. Entreprise II thus provides various tools for simulation, prototyping, specification, design, coding, testing, maintenance, etc. within one project or for different projects in the same IPSE.



Vertical tools in the IPSE

5 - ENTREPRISE II - AN OPEN INTEGRATOR OF SOFTWARE ENGINEERING SOLUTIONS

Entreprise II is the ideal environment for developing medium- and large-size software.applications It is typically used for projects involving 100 000 or more lines of code, where the standard activities of specification, design, coding and testing account for only 30% of the total cost, the rest being accounted for by project, configuration, documentation and maintenance management.

Software maintenance accounts for more than 50% of the total cost, so integrating horizontal tools used for configuration, documentation and project management with the standard vertical tools used for specification, coding, etc.results in a 30% increase in productivity starting from the second project. The vertical tools improve the performance of individuals in their particular production activity, while the complete, integrated Entreprise II environment works to meet the goal of increased productivity on a higher level, i.e. that of collective performance.

Entreprise II is an open integrator of CASE solutions. The three companies involved in the developement of Entreprise II (SYSECA, CR2A and STERIA) have set up a new company, EII Software which is in charge of marketing the product. This company also has for aim to unite all CASE tools manufacturers and retailers working together with Entreprise II to give their customers new levels of productivity, security and long product life.

In the USA, ALSYS Inc. is distributing Entreprise II under the name of FreedomWorks™.

TRADEMARKS

ARTEMIS is a registered trademark of Lucas Management Systems
ENTREPRISE is a registered trademark of The Délégation Générale pour l'Armement
FRAMEMAKER is a registered trademark of Frame Technology
FREEDOMWORKS is a registered trademark of Alsys Inc.
MOTIF is a registered trademark of OSF (Open Software Foundation)
OPEN LOOK is a registered trademark of AT&T
TPS is a registered trademark of Interleaf
UNIX is a registered trademark of AT&T
X. WINDOW is a registered trademark of MIT (Massachusetts Institute of Technology)

Discussion

Question F. CHERATZU

What can you say about NAPI's (North American PCTE Initiative) intention of building and distributing for free an integration platform based on ECMA PCTE?

Reply

As far as I know, but I may not be the right person to answer, it seems to appear that the objective of having a NAPI public implementation of ECMA PCTE is no more pursued. NAPI charter is not yet completely established, so that it must be checked.

Question R. SZYMANSKI

1. Does the lack of a validation suite for PCTE hamper tool production by the vendor?

2. Which version of the PCTE standard do you use?

Reply

1. A PCTE 1.5 validation suite has been used for Entreprise II validation by the french MoD. There are intents in the North American PCTE Initiative to set up a ECMA PCTE validation suite. Intents do also exist within the CEC.

2. The initial version of Entreprise II is based on PCTE 1.5. Entreprise II will migrate very quickly to ECMA PCTE.

Question C. BENJAMIN

When integrating a commercial tool into Entreprise II, do you have to do some software modification?

Reply

Two integration models are possible :

- a tight integration, where the the tool uses directly the Entreprise II Framework services. In that case, that tool must be modified;

- a loose integration, where a "capsule" is built around the tool, that supports all the interface with the framework and its repository.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-CP-545	ISBN 92-835-0725-8	UNCLASSIFIED/ UNLIMITED

5. Originator Advisory Group for Aerospace Research and Development
North Atlantic Treaty Organization
7 Rue Ancelle, 92200 Neuilly sur Seine, France

6. Title AEROSPACE SOFTWARE ENGINEERING FOR ADVANCED SYSTEMS ARCHITECTURES

7. Presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

8. Author(s)/Editor(s)	9. Date
Various	November 1993
10. Author's/Editor's Address	**11. Pages**
Various	352

12. Distribution Statement There are no restrictions on the distribution of this document. Information about the availability of this and other AGARD unclassified publications is given on the back cover.

13. Keywords/Descriptors

Software	Programing
ADA	Software management
Object oriented design	Software environments
Software design	Artificial intelligence
Software specification	Avionics
Software validation & testing	Aerospace

14. Abstract

During the past decade, many avionics functions which have traditionally been accomplished with analogue hardware technology are now being accomplished by software residing in digital computers. Indeed, it is clear that in future avionics systems, most of the functionality of an avionics system will reside in software. In order to design, test and maintain this software, software development/support environments will be extensively used. The significance of this transition to software is manifested in the fact that 50 percent or more of the cost of acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered with the following representing a partial set of topics: Aerospace Electronics Software Specification, Software Design, Programming Practices and Techniques, Software Validation and Testing, Software Management and Software Environments.

AGARD Conference Proceedings 545
Advisory Group for Aerospace Research and Development, NATO
AEROSPACE SOFTWARE ENGINEERING FOR ADVANCED SYSTEMS ARCHITECTURES
Published November 1993
352 pages

During the past decade, many avionics functions which have traditionally been accomplished with analogue hardware technology are now being accomplished by software residing in digital computers. Indeed, it is clear that in future avionics systems, most of the functionality of an avionics system will reside in software. In order to design, test and maintain this software, software development/support environments will be extensively used. The significance of this transition to software is manifested in the fact that 50 percent or more of the cost of

P.T.O.

AGARD-CP-545

Software
ADA
Object oriented design
Software design
Software specification
Software validation & testing
Programing
Software management
Software environments
Artificial intelligence
Avionics
Aerospace

AGARD Conference Proceedings 545
Advisory Group for Aerospace Research and Development, NATO
AEROSPACE SOFTWARE ENGINEERING FOR ADVANCED SYSTEMS ARCHITECTURES
Published November 1993
352 pages

During the past decade, many avionics functions which have traditionally been accomplished with analogue hardware technology are now being accomplished by software residing in digital computers. Indeed, it is clear that in future avionics systems, most of the functionality of an avionics system will reside in software. In order to design, test and maintain this software, software development/support environments will be extensively used. The significance of this transition to software is manifested in the fact that 50 percent or more of the cost of

P.T.O.

AGARD-CP-545

Software
ADA
Object oriented design
Software design
Software specification
Software validation & testing
Programing
Software management
Software environments
Artificial intelligence
Avionics
Aerospace

AGARD Conference Proceedings 545
Advisory Group for Aerospace Research and Development, NATO
AEROSPACE SOFTWARE ENGINEERING FOR ADVANCED SYSTEMS ARCHITECTURES
Published November 1993
352 pages

During the past decade, many avionics functions which have traditionally been accomplished with analogue hardware technology are now being accomplished by software residing in digital computers. Indeed, it is clear that in future avionics systems, most of the functionality of an avionics system will reside in software. In order to design, test and maintain this software, software development/support environments will be extensively used. The significance of this transition to software is manifested in the fact that 50 percent or more of the cost of

P.T.O.

AGARD-CP-545

Software
ADA
Object oriented design
Software design
Software specification
Software validation & testing
Programing
Software management
Software environments
Artificial intelligence
Avionics
Aerospace

acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered with the following representing a partial set of topics: Aerospace Electronics Software Specification, Software Design, Programming Practices and Techniques, Software Validation and Testing, Software Management and Software Environments.

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

ISBN 92-835-0725-8

acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered with the following representing a partial set of topics: Aerospace Electronics Software Specification, Software Design, Programming Practices and Techniques, Software Validation and Testing, Software Management and Software Environments.

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

ISBN 92-835-0725-8

acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered with the following representing a partial set of topics: Aerospace Electronics Software Specification, Software Design, Programming Practices and Techniques, Software Validation and Testing, Software Management and Software Environments.

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

ISBN 92-835-0725-8

acquiring and maintaining advanced weapons systems is directly related to software considerations. It is also significant that this dependence on software provides an unprecedented flexibility to quickly adapt avionics systems to changing threat and mission requirements. Because of the crucial importance of software to military weapons systems, all NATO countries are devoting more research and development funds to explore every aspect of software science and practice.

The purpose of this Symposium was to bring together military aerospace software experts from all NATO countries to share the results of their software research and development and virtually every aspect of software was considered with the following representing a partial set of topics: Aerospace Electronics Software Specification, Software Design, Programming Practices and Techniques, Software Validation and Testing, Software Management and Software Environments.

Papers presented at the Avionics Panel Symposium held in Paris, France, 10th—13th May 1993.

ISBN 92-835-0725-8

AGARD

NATO OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE

FRANCE

Télécopie (1)47.38.57.99 · Télex 610 176

DIFFUSION DES PUBLICATIONS AGARD NON CLASSIFIEES

Aucun stock de publications n'a existé à AGARD. A partir de 1993, AGARD détiendra un stock limité des publications associées aux cycles de conférences et cours spéciaux ainsi que les AGARDographies et les rapports des groupes de travail, organisés et publiés à partir de 1993 inclus. Les demandes de renseignements doivent être adressées à AGARD par lettre ou par fax à l'adresse indiquée ci-dessus. *Veuillez ne pas téléphoner.* La diffusion initiale de toutes les publications de l'AGARD est effectuée auprès des pays membres de l'OTAN par l'intermédiaire des centres de distribution nationaux indiqués ci-dessous. Des exemplaires supplémentaires peuvent parfois être obtenus auprès de ces centres (à l'exception des Etats-Unis). Si vous souhaitez recevoir toutes les publications de l'AGARD, ou simplement celles qui concernent certains Panels, vous pouvez demander à être inclu sur la liste d'envoi de l'un de ces centres. Les publications de l'AGARD sont en vente auprès des agences indiquées ci-dessous, sous forme de photocopie ou de microfiche.

CENTRES DE DIFFUSION NATIONAUX

ALLEMAGNE
Fachinformationszentrum,
Karlsruhe
D-7514 Eggenstein-Leopoldshafen 2

BELGIQUE
Coordonnateur AGARD-VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Directeur du Service des Renseignements Scientifiques
Ministère de la Défense Nationale
Ottawa, Ontario K1A 0K2

DANEMARK
Danish Defence Research Establishment
Ryvangs Allé 1
P.O. Box 2715
DK-2100 Copenhagen Ø

ESPAGNE
INTA (AGARD Publications)
Pintor Rosales 34
28008 Madrid

ETATS-UNIS
National Aeronautics and Space Administration
Langley Research Center
M/S 180
Hampton, Virginia 23665

FRANCE
O.N.E.R.A. (Direction)
29, Avenue de la Division Leclerc
92322 Châtillon Cedex

GRECE
Hellenic Air Force
Air War College
Scientific and Technical Library
Dekelia Air Force Base
Dekelia, Athens TGA 1010

ISLANDE
Director of Aviation
c/o Flugrad
Reykjavik

ITALIE
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
Aeroporto Pratica di Mare
00040 Pomezia (Roma)

LUXEMBOURG
Voir Belgique

NORVEGE
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
N-2007 Kjeller

PAYS-BAS
Netherlands Delegation to AGARD
National Aerospace Laboratory NLR
P.O. Box 90502
1006 BM Amsterdam

PORTUGAL
Força Aérea Portuguesa
Centro de Documentação e Informação
Alfragide
2700 Amadora

ROYAUME UNI
Defence Research Information Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

TURQUIE
Milli Savunma Başkanlığı (MSB)
ARGE Daire Başkanlığı (ARGE)
Ankara

Le centre de distribution national des Etats-Unis (NASA/Langley) ne détient PAS de stocks des publications de l'AGARD. D'éventuelles demandes de photocopies doivent être formulées directement auprès du NASA Center for Aerospace Information (CASI) à l'adresse suivante:

AGENCES DE VENTE

NASA Center for
Aerospace Information (CASI)
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
United States

ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris
France

The British Library
Document Supply Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
Royaume Uni

Les demandes de microfiches ou de photocopies de documents AGARD (y compris les demandes faites auprès du CASI) doivent comporter la dénomination AGARD, ainsi que le numéro de série d'AGARD (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Veuiller noter qu'il y a lieu de spécifier AGARD-R-nnn et AGARD-AR-nnn lors de la commande des rapports AGARD et des rapports consultatifs AGARD respectivement. Des références bibliographiques complètes ainsi que des résumés des publications AGARD figurent dans les journaux suivants:

Scientifique and Technical Aerospace Reports (STAR)
publié par la NASA Scientific and Technical
Information Program
NASA Headquarters (JTT)
Washington D.C. 20546
Etats-Unis

Government Reports Announcements and Index (GRA&I)
publié par le National Technical Information Service
Springfield
Virginia 22161
Etats-Unis

(accessible également en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

Imprimé par Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ