NASA
CP
2064
c.1

NASA Conference Publication 2064

LOAN COPY: RETU
AFWL TECHNICAL L
KIRTLAND AFB, N

0067321

TECH LIBRARY KAFB, NM

Tools for Embedded Computing Systems Software

Preprint for a workshop
held in Hampton, Virginia
November 7-8, 1978

NASA

TECH LIBRARY KAFB, NM

0067321

NASA Conference Publication 2064

Tools for Embedded Computing Systems Software

Preprint for a workshop sponsored by NASA Langley Research Center, Hampton, Virginia, and the American Institute of Aeronautics and Astronautics, New York, and held in Hampton, Virginia, November 7-8, 1978

NASA
National Aeronautics and Space Administration

Scientific and Technical Information Office

1978

FOREWORD

NASA, in cooperation with the AIAA Computer Systems Technical Committee, sponsored this workshop on Tools for Embedded Computing Systems Software. The rapidly increasing capabilities and decreasing costs of digital computing systems have meant orders of magnitude increases in the uses of these systems in aerospace systems, particularly onboard satellites and aircraft. Its labor intensive nature and crucial role in proper operation of the embedded computing system have made the development and testing of software an expensive and critical function. To cut the cost of the development of the embedded system software while enhancing its reliability, a number of tools (i.e., computer programs) have been developed in the recent past.

The objectives of this workshop were to assess the current state of these tools and determine directions for future tool development. The workshop was organized around four major areas: Tools and the Software Environment (development and testing); Tools and Language Processors; Tools and Software Requirements, Design, and Specification; and Tools and Verification and Validation (analysis and testing). This document contains only a synopsis of the talk and the key figures of each formal workshop presentation together with summaries by each of the session chairmen.

The synopses were submitted as camera-ready copy prior to the workshop. Only minor editorial changes have been made and a title page and abstract have been added. The assistance of the Scientific and Technical Information Programs Division of the NASA Langley Research Center in publishing this preprint is gratefully acknowledged.

The workshop was structured to provide ample time for audience interaction in addition to the formal presentations. On the evening of November 7, a panel discussion on "Software Management, Methodology, and Tools" was held. The panelists were Victor Basili, University of Maryland; Jack Garman, Manager of Shuttle Avionics Software at NASA Johnson Space Flight Center; James Stringer, Computer Sciences Corporation; and Kenneth A. Hales, Boeing Aerospace Company.

The Workshop Subcommittee were Terry A. Straeter, Chairman, NASA Langley Research Center; Phil S. Babel, Wright-Patterson Air Force Base; James W. Clark, United Technologies Research Center; George R. Fath, General Electric Company; Charles H. Fletcher, Stromberg Carlson Corporation; Sabina H. Saib, General Research Corporation; Robert J. Schwartz, McDonnell Douglas Astronautics Company; and Lynn S. Wilson, Grumman Data Systems Corporation.

Use of manufacturers or identification of commercial products in this report does not constitute an official endorsement of such manufacturers or products, either expressed or implied, by the National Aeronautics and Space Administration.

Terry A. Straeter
Program Chairman

CONTENTS

FOREWORD iii

SESSION IA - TOOLS AND THE SOFTWARE ENVIRONMENT (DEVELOPMENT)
Chairman: Kenneth A. Hales, Boeing Aerospace Company

1. THE SOFTWARE ENVIRONMENT - NOW AND IN THE FUTURE 1
Kenneth A. Hales, Boeing Aerospace Company

2. THE SYSTEM DESIGN LABORATORY (SDL) 7
W. Lin Sutton and Patricia Santoni, Naval Ocean Systems Center

3. INSTRUMENTATION AND CONTROL OF A VIRTUAL MACHINE 11
Majorie K. Kirchoff and S. Harris Dalrymple, McDonnell Douglas Astronautics Company

4. NASA ISIS 15
W. Joseph Berman, University of Virginia

5. SOFTWARE SYSTEMS DEVELOPMENT AT GRUMMAN AEROSPACE CORPORATION 21
Jack Rosenbaum, Grumman Aerospace Corporation

SESSION II - TOOLS AND LANGUAGE PROCESSORS
Chairman: Fred H. Martin, Intermetrics, Inc.

6. LANGUAGE TOOLS - WHERE THE LEVERAGE IS 25
Fred H. Martin, Intermetrics, Inc.

7. PATH EXPRESSIONS FOR REAL-TIME PROGRAMMING 29
R. H. Campbell, University of Illinois at Urbana-Champaign

8. VERIFIABLE PASCAL 33
Sabina H. Saib, General Research Corporation

9. A NEW META ASSEMBLER 37
K. V. Smith, Z. Jelinski, J. B. Churchwell, and S. Park, McDonnell Douglas Astronautics Company-West

10. A UNIVERSAL FLOWCHARTER 41
J. Rood, T. To, and D. Harel, Higher Order Software, Inc.

11. THE APPLICATION OF SOFTWARE ENGINEERING TECHNIQUES TO THE DESIGN OF RELATIVELY MACHINE-INDEPENDENT CODE GENERATORS 45
Robert E. Noonan, College of William and Mary, and Patricia Timpanaro, Computer Sciences Corporation

SESSION IB - TOOLS AND THE SOFTWARE ENVIRONMENT (TESTING)
Chairman: Kenneth A. Hales, Boeing Aerospace Company

12. AN AVIONICS SOFTWARE DEVELOPMENT EXPERIENCE 49
L. C. Klos, General Dynamics Corporation

13. FUNCTIONAL SIMULATION OF SPACE SHUTTLE FLIGHT PROGRAMS 53
Arra Avakian, Intermetrics, Inc.

14. PROVE - A TOOL FOR SOFTWARE VERIFICATION 57
Randall J. Varga, The Singer Company, Kearfott Division

15. INTERPLANETARY SPACECRAFT COMPUTER SOFTWARE TEST AND VALIDATION TOOLS 65
Daniel E. Erickson, Jet Propulsion Laboratory

16. A SOFTWARE CHANGE DEVELOPMENT LABORATORY FOR SUPPORTING AN AGGREGATE OF EMBEDDED COMPUTER SYSTEMS 71
F. H. Kishi, TRW Defense and Space Systems Group, and
D. R. Corder, Oklahoma City Air Logistics Center

SESSION III - TOOLS AND SOFTWARE REQUIREMENTS, DESIGN, AND SPECIFICATIONS
Chairman: Lt. Col. Charles John Grewe, Jr., U.S. Air Force
Electronics Systems Division

17. TOOLS AND SOFTWARE REQUIREMENTS, DESIGN, AND SPECIFICATIONS 75
Lt. Col. Charles John Grewe, Jr., U.S. Air Force
Electronics Systems Division

18. THE DEVELOPMENT OF PROGRAMS TO MEET EXCEPTIONAL RELIABILITY REQUIREMENTS 81
P. M. Melliar-Smith, SRI International

19. DESCRIBING A TRIPLY-REDUNDANT FLIGHT CONTROL SYSTEM FOR VERIFICATION OF DESIGN 85
Edward R. Rang, Honeywell, Inc.

20. SOFTWARE DESIGNERS WORKBENCH - REQUIREMENTS AND DESIGN TOOLS FOR EXPRESSION AND EVALUATION 89
Paul A. Scheffer, Martin Marietta Aerospace

21. TOOLS FOR EMBEDDED SOFTWARE DESIGN VERIFICATION 93
J. C. Enos and R. R. Willis, Hughes Aircraft Company

22. AN EVENT-BASED DESIGN METHOD FOR EMBEDDED SOFTWARE SYSTEMS 97
William E. Riddle, University of Colorado at Boulder

23. A TECHNIQUE TO AID IN THE DESIGN AND ANALYSIS OF DYNAMICALLY STRUCTURED, CONCURRENT SOFTWARE SYSTEMS 101
Jack C. Wileden, University of Massachusetts

SESSION IVA - TOOLS AND VERIFICATION AND VALIDATION (ANALYSIS)
Chairman: Sabina H. Saib, General Research Corporation

24. USING HDM FOR EMBEDDED SYSTEMS 105
L. Robinson and K. N. Levitt, SRI International

25. AN INTEGRATED VERIFICATION AND VALIDATION TOOL FOR FLIGHT SOFTWARE . . 109
Richard N. Taylor, Leon J. Osterweil, and Leon G. Stucki,
Boeing Computer Services Company

26. USE OF SYMBOLIC EXECUTION IN VERIFICATION AND VALIDATION 113
Marilyn S. Fujii and Michael A. Ikezawa, Logicon, Inc.

27. SQLAB - TOOLS FOR PROGRAM VERIFICATION 117
Sabina H. Saib, General Research Corporation

28. A SOFTWARE QUALITY ASSURANCE EXPERIMENT 121
J. P. Benson and S. H. Saib, General Research Corporation

SESSION IVB - TOOLS AND VERIFICATION AND VALIDATION (TESTING)
Chairman: Sabina H. Saib, General Research Corporation

29. A SIMULATOR DEVICE FOR VALIDATION OF GENERAL AVIONICS
EMBEDDED SOFTWARE 125
Byron M. Allen and Gary H. Barber, Intermetrics, Inc.

30. DYNAMIC INTEGRATED TEST FOR THE SPACE SHUTTLE 129
Saul F. Stanten, Intermetrics, Inc.

THE SOFTWARE ENVIRONMENT - NOW AND IN THE FUTURE

Kenneth A. Hales
Boeing Aerospace Company

SUMMARY

The present status of facilities and support tools used to develop and integrate software are characterized and described. An assessment of these present systems is made and a projection is offered which characterizes software development systems of the future.

PRESENT SYSTEM CHARACTERISTICS

Software Development Triangle

The design and development of embedded software systems is performed on computers found in software development laboratories. The selection of the computers which contain the embedded software code is a function of the operational requirements, as well as a function of the languages used to develop software, and the types of computers used to initially generate the software. Figure 1 describes the software development triangle. In the bottom left-hand corner of figure 1, the program generation center computers are identified. These computers are called "host computers." They provide a home for the support software used in conjunction with the application code to generate and test software which will ultimately run on the operational or "target" computers shown in the bottom right-hand corner of figure 1. To complete the triangle, higher-order languages are identified.

The triangle is subjected to external influences. For example, the higher-order language of the system of software is influenced by the desire to have high programmer productivity and cost-effective software maintainability, and is a function of specified government regulations. The target computers are strongly influenced by operational requirements. The type of program generation center computers is influenced by the quantity of software that has to be generated. For example, if only a small amount of software has to be generated, it is possible that the target computers can be used to provide adequate host support.

In selecting the host computers, target computers, and the higher-order language for the embedded system under consideration, availability of compilers, as well as a complement of extensive support software, must be considered in making the final decision. The ideal situation is to select target computers which are highly satisfactory in meeting operational requirements while also being compatible with program generation center computers which contain an extensive suite of support software. Ideally also, is the selection of a program generation center suite of computers which can be dedicated to the specific project or is available within the company's central computing facility run by a central/trained staff. The program generation center should also be compatible with the most effective means of generating software, which in some cases, requires the ability to code interactively in a higher-order language.

To effectively integrate software elements into a working system, extensive amounts of simulations are performed. Figure 1 indicates that host simulations are used within the program generation center. Host software accepts functional simulations, or instruction level simulations, so that the developed software may be extensively integrated against models of the external environment while still operating on the host computers. Real time simulations are also used. These simulations require software, computers, and possibly hardware, and interface with the operational target computers. To properly generate a solution to the problem of the most effective software development system, the aspects of the software development triangle must be comprehensively evaluated.

Software Development and Integration Facilities

Figure 2 describes the ingredients of a successful environment. Figure 2(a) symbolizes a building or a facility which is designed for the programmer. Further material is available on the architectural design of a laboratory for program development (reference 1). Within a building, a facility will exist which will contain a host machine with extensive support software and appropriate peripherals (see figure 2(b)). This computer complex is used to develop the software. The software project may also contain a facility which has a target computer interfacing with a simulation computer and selected pieces of system hardware. The latter facility is used to integrate the software. Figure 2(c) symbolizes this target machine complex. The target computer runs the embedded application software. The simulation computer contains a model of the external environment. The hardware is representative of one or more pieces of hardware which the target computer must control. Initially, the target computer will work directly and solely with the simulation computer. As hardware becomes available during the development of the total project, pieces of mature hardware can be entered into the target machine complex and simulated elements contained within the simulation computer dropped out. This will allow the target computer software to be integrated as a system and checked out against the models of the external hardware and high confidence pieces of the actual computer hardware prior to releasing the target computer and software to the system-oriented software/hardware integration phases of a project. Figure 2(d) symbolizes the final aspects of the ingredients of a successful environment. Here tools, standards, training, etc., are shown which are project unique and company unique. These products have been built with the productive development and management of software in mind.

The host machine complex and target machine complex are sometimes combined, or sometimes separate in a given application. Figure 3 shows the most common combinations of computer elements. Figure 3(a) shows a multipurpose host complex where the host computer is equivalent to the machine which will be delivered as part of the embedded computing system. Here the host machine complex is used to develop the software, as well as integrate the software. The host machine will contain functional simulations, as well as real-time simulations so that software may be generated, checked out functionally, and then operated in real time against a real-time simulation of the external environment contained within itself. In the case that the target computer is quite different from the host machine computer, a software development and integration complex is created (see figure 3(b)). The host machine complex is used to develop software and contains a real-time simulation which interfaces with the target computer. The target machine software is exercised against the real-time simulation in the host machine complex. The most extensive software development facility contains a partitioned software development and integration complex (figure 3(c)). The host machine complex shown in the figure is the host for software development. The second portion of the facility is a software integration center consisting of the target computer, a simulation computer, and selected external hardware. The software integration center is used to exercise target machine software. Numerous examples of various applications of the combinations shown in figure 3 are found in the literature.

ASSESSMENT OF PRESENT SYSTEMS

The Computer Technology Forecast and Weapon Systems Impact Study was held at the U.S. Air Force Academy from August 14-25, 1978. The Technology Forecast and Impact Study made an assessment of the present state of the art in programming environment. The panel of experts related to software stated that exaggerated claims have been and continue to be made for isolated tools and techniques. They pessimistically stated that most program development is done with severely inadequate tools. They also indicated that a compiler is frequently equated with a programming environment. In reality, a compiler constitutes only one small, albeit important, component of an automated environment for programming. They made the point that management awareness of the progress of software development is woefully inadequate. The study also confirmed that experimental use is being made of selective tool and technique concepts not yet widely available. In short, the study panel indicated that the trends are in the right direction, but significant progress has to be made before the numerous claims for techniques to enhance productivity will become a reality.

DESCRIPTION OF FUTURE SYSTEMS

Future host systems will consist of a generalized input/output port accessible to the user. This port will interface with remote, distributed systems of computers as shown in figure 4(a). The host system of software will exist on numerous computers. The user will be able to program in his choice of higher order languages. Extensive support software systems for numerous target machines, including microprocessors, will exist. The host system will include extensive development tools. Algorithm banks will exist on the host system with automated language/machine dependent translation capabilities so that the algorithm banks are portable between applications. Evidence of trends toward the future host system is described in reference 2. In addition, the Technology Forecast and Impact Study describes the National Software Works (NSW). This concept is a joint Air Force/ARPA sponsored program to provide a configurable programming environment through the use of computer networking. It allows the software developer to use software tools on various computer systems.

Future integration and maintenance systems will consist of a generalized host complex, as well as a generalized target/simulation computer complex. This concept is proposed in reference 3 and symbolized in figure 4(b).

The Technology Forecast and Impact Study forecasts that future trends will see generalized programming environments being designed and selectively implemented. Hardware manufacturers will provide machine and language dependent environments as subelements of the generalized programming environment. Techniques will be developed to isolate language and operating system dependencies as much as possible. They pointed out that the higher order language standardization within DOD will make it possible to achieve a rich program development environment accessible to a large number of people. The generalized programming environment will host more and more automation tools which will be provided to enhance the development, management and testing of future systems.

Future directions are becoming more focused. Future systems are now able to be articulated and described. Now all that is needed is effort, time, and commitment.

REFERENCES

1. McCue, Gerald M.: IBM's Santa Teresa Laboratory - Architectural Design for Program Development. IBM Syst. J., vol. 17, no. 1, 1978, pp. 4-25.

2. Sutton, W. Lin; and Santoni, Patricia: The System Design Laboratory (SDL). Tools for Embedded Computing Systems Software, NASA CP-2064, 1978. (Paper no. 2 of this compilation.)

3. Kishi, F. H.; and Corder, D. R.: A Software Change Development Laboratory for Supporting an Aggregate of Embedded Computer Systems. Tools for Embedded Computing Systems Software, NASA CP-2064, 1978. (Paper no. 16 of this compilation.)

The Software Development Triangle

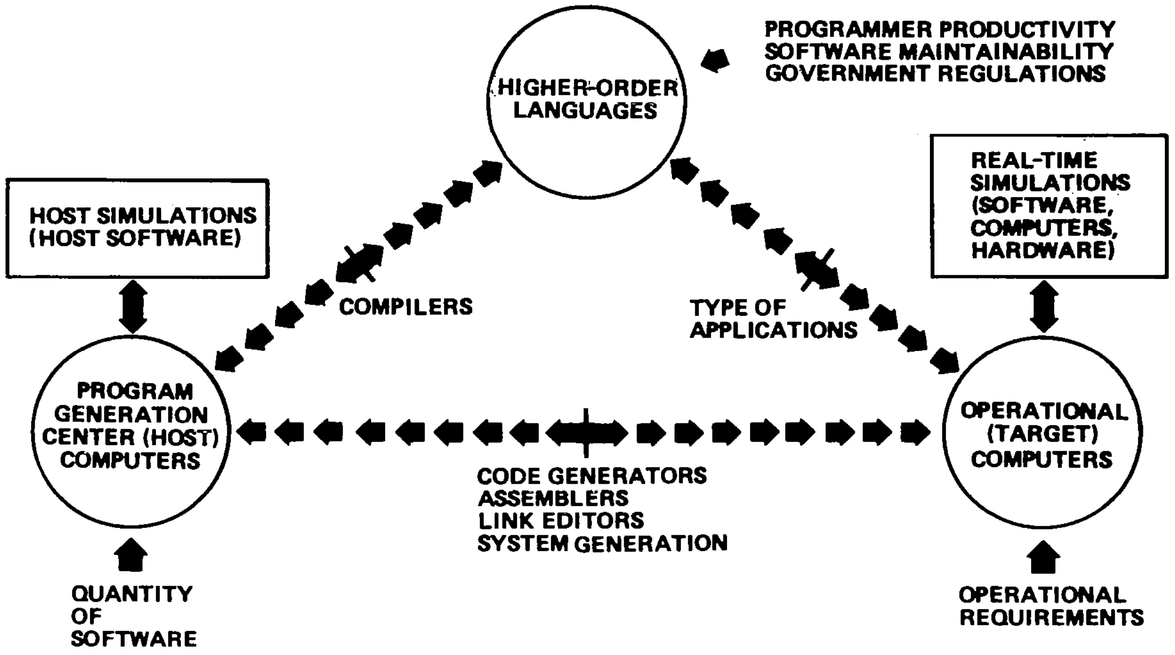
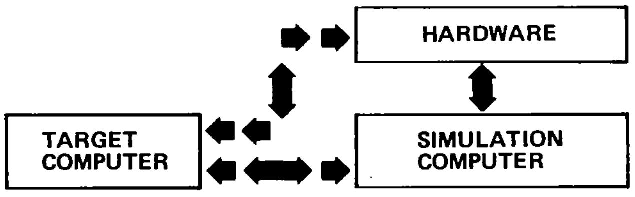


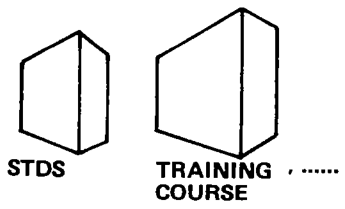
Figure 1

Ingredients of a Successful Environment

(a) Facility designed for programmer. (b) Host machine with support software and appropriate peripherals.



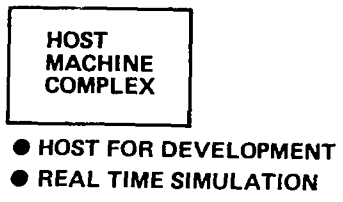
(c) Target machine complex.



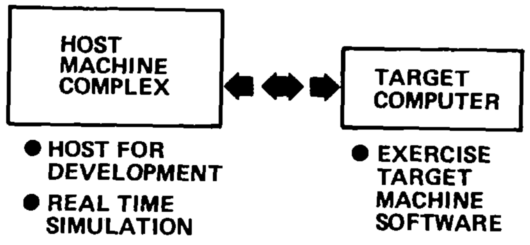
(d) Tools, standards, and training with the productive development/management of software in mind.

Figure 2

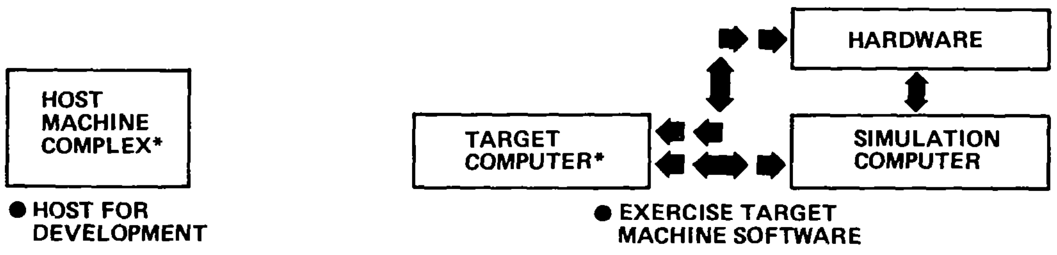
Most Common Combinations of Computer Elements



(a) Multipurpose host complex with host ≈ target.



(b) Software development and integration complex.

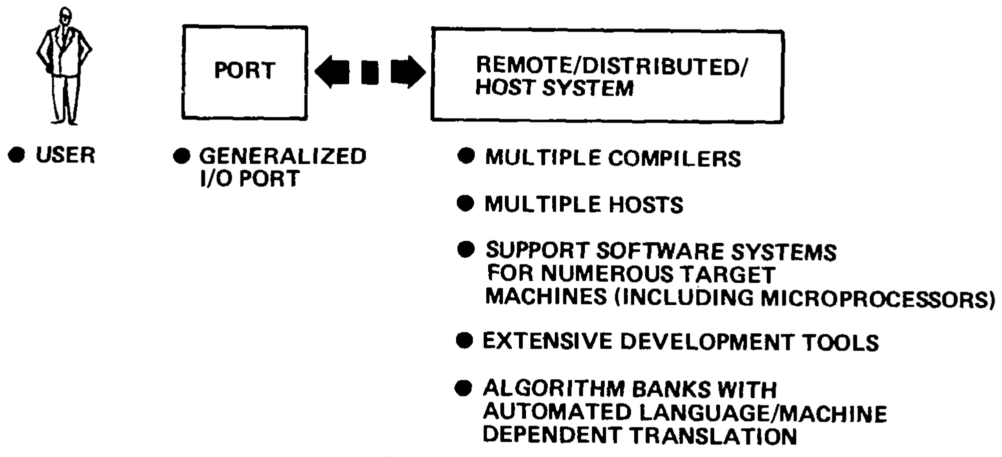


* IN SOME CASES THE TARGET COMPUTER IS USED AS THE HOST AS WELL

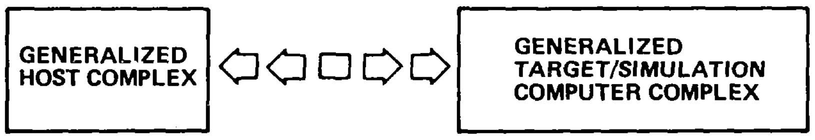
(c) Partitioned software development and integration complex.

Figure 3

Future Systems


PORT
REMOTE/DISTRIBUTED/ HOST SYSTEM
● USER
● GENERALIZED I/O PORT
● MULTIPLE COMPILERS
● MULTIPLE HOSTS
● SUPPORT SOFTWARE SYSTEMS FOR NUMEROUS TARGET MACHINES (INCLUDING MICROPROCESSORS)
● EXTENSIVE DEVELOPMENT TOOLS
● ALGORITHM BANKS WITH AUTOMATED LANGUAGE/MACHINE DEPENDENT TRANSLATION

(a) Future hosts.



(b) Future integration and maintenance systems.

Figure 4

THE SYSTEM DESIGN LABORATORY (SDL)

W. Lin Sutton and Patricia Santoni
Naval Ocean Systems Center

The System Design Laboratory (SDL) is a joint ARPA/Navy project which brings together in one widely accessible, cohesive environment the tools which are needed by the designers and developers of embedded computer systems. It is intended to support work in software, firmware, and hardware systems design and development. The goal of SDL is two-fold: (1) to make those tools that exist available to the designers of Navy Systems and (2) to promote the necessary research in areas where adequate tools do not exist.

Eventually, SDL will include tools in the areas of requirements (e.g., Hierarchical Development Methodology hierarchy manager and Specification and Assertion Language (SPECIAL) analyzer), modelling (e.g., simulation languages), implementation (e.g., language processors, emulator/debuggers for target machines), and testing (e.g., the Automated Testing Analyzer for CMS-2M). In addition to these, there will be tools to support text composition, document generation according to MIL standards, project management, and the various reformatting routines that may be necessary to prepare the output of one tool for input to another tool.

Currently, SDL primarily offers an AN/UYK-20 and Intel 8080 software generation center. The outstanding feature in this environment is a microprogrammed emulation of each of these machines which provides a CPU and peripherals with all of the options available on the real equipment, plus very sophisticated debugging capabilities. These tools reside on the ARPANET on two host machines, (a DEC PDP-10 and an IBM S/360-91) and a micro-programmable emulator, the MLP-900. All processing is done on the host machines to generate code for the target machines. The user primarily works from a terminal until all possible testing has been done via the SDL facility and it is time to generate a tape of his system to try on his actual hardware.

Since the opening of the SDL/IOC, several projects from different Navy and industry activities have made use of its facilities. Among them have been projects from NESEC, NOSC, Litton, and SDC which primarily made use of the AN/UYK-20 software generation and debugging capabilities; projects from FCDSSA and NOSC which have made use of the PSL/PSA and URL/URA tools; and projects from NOSC which have experimented with the upcoming HDM tools, including the specification language SPECIAL. The usage to date has been primarily experimental and has served to shake out many of the SDL concepts and tools. Users have shown special enthusiasm for those tools which are not available elsewhere, including the PRIM emulation tools, the HDM tools, and the ATA.

The SDL is an idea which is long overdue in the DOD. In the future it will continue to evolve and include more of the sort of tools indicated above and pursue more of the research needed to bring the cost of military embedded computer systems down to a more reasonable level.

10C (INITIAL OPERATING CAPABILITY)

CMS-2 SOFTWARE DEVELOPMENT FACILITY

UYK-20 SOFTWARE DEVELOPMENT FACILITY

TOOLS ON HOST COMPUTERS IN THE ARPANET

NSW (NATIONAL SOFTWARE WORKS) INTERFACE AS AVAILABLE

TOOL SET (FOR UYK-20 AND INTEL 8080)

- CROSS-COMPILERS
- CROSS-ASSEMBLERS
- EMULATOR/SIMULATOR

Figure 1

AUTOMATED DESIGN FOR C^3 SYSTEMS

SYSTEM DESIGN LAB (SDL)

Transitions (Users)

NESEC, San Diego, use of UYK-20 emulation/debug tool during test, acceptance, and maintenance of NAVMACS

FCDSSA, San Diego and NOSC utilizing user requirements language/user requirements analyzer (URL/URA) in design of new NTDS program

NOSC using Hierarchical Design Methodology (HDM) to partition design for GPS navigation system user terminal

NOSC JTIDS project using the UYK-20 emulation and debug capabilities in the development of the UYK-20 netway and biway software

Litton using SDL tools in development of Advanced Communications Control System (ACCS) software

Figure 2

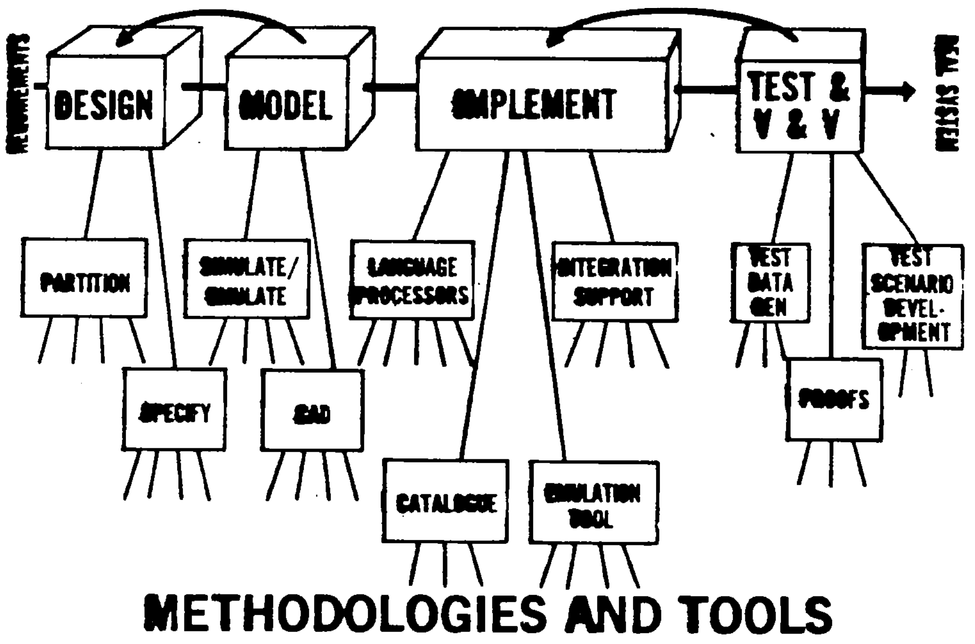


Figure 3

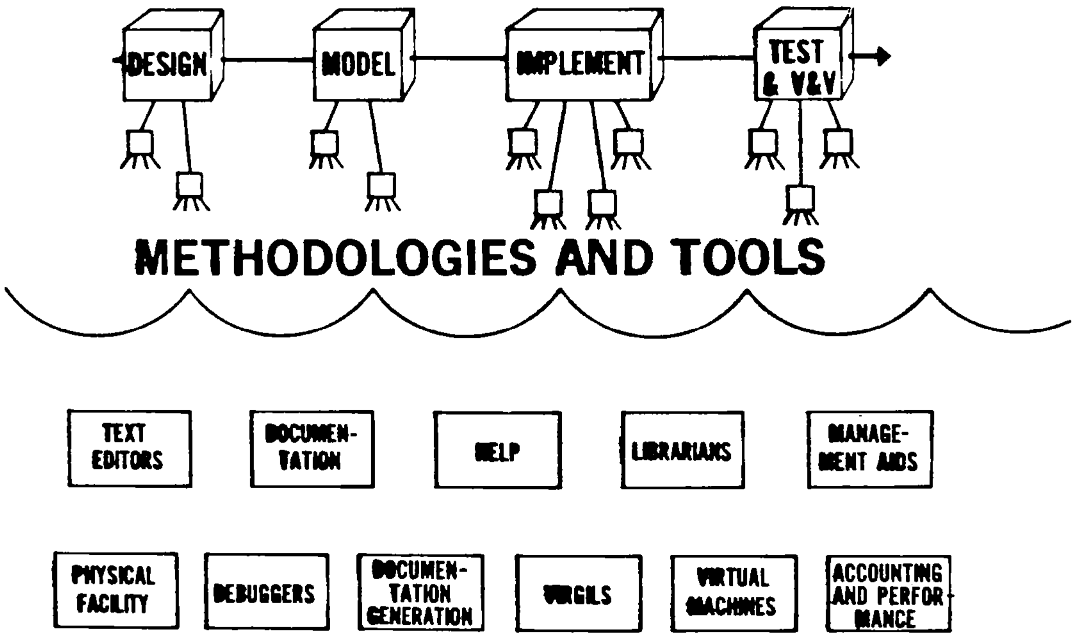


Figure 4

APPROACH:

1. DEVELOP A REPOSITORY TO HOST DESIGN AND DEVELOPMENT TOOLS
2. PERFORM FEASIBILITY DEMONSTRATIONS OF NEW TECHNOLOGIES. MAKE TOOLS WIDELY AVAILABLE IN ACCORDANCE WITH MERIT
3. SPONSOR NEW RESEARCH IN DESIGN TECHNOLOGIES WHERE NOT OTHERWISE PROVIDED

Figure 5

PURPOSE:

UPGRADE THE TECHNOLOGY AVAILABLE TO DESIGNERS AND DEVELOPERS OF EMBEDDED MILITARY COMPUTER SYSTEMS

Figure 6

INSTRUMENTATION AND CONTROL OF A VIRTUAL MACHINE

Majorie K. Kirchoff and S. Harris Dalrymple
McDonnell Douglas Astronautics Company

The use of microcoded emulations as a technique for the development and validation of real-time software has proven to be a very versatile and effective tool. Since the virtual machine produced through emulation is a product of software, it is possible to imbed a wide range of debugging facilities not ordinarily available in its hardware counterpart. Extensive error checking may be performed, programming standards and application peculiar conditions monitored, and performance measurement statistics generated. Also, external interfaces may be simulated to provide a dynamic execution environment as well as collect output such as a telemetry stream for postmortem analyses.

To control this complex virtual machine, a unique dual emulation approach has been developed which capitalizes upon the availability of a wide spectrum of support software on the secondary machine. Hosting the emulation systems is a Nanodata QM-1, a machine specifically designed to support multiple emulations. It features a three-level memory system, each of which is writable: Main Store (core) for target machine code, Control Store (LSI) for vertical microcode, and Nanostore (LSI) for horizontal microcode.

By combining a NOVA 1200 emulator with the real-time computer emulator, software development systems for three control processors have been developed: SCP-234, MAGIC 352, and LC-4516. While each of these emulators is quite unique, their control has been uniformly accomplished by means of programs, written primarily in FORTRAN, running on the emulated NOVA. The memory resources of the QM-1 are divided between the two emulators while the peripheral devices are assigned to the NOVA.

Communicaion between the emulators is accomplished via a common buffer residing in control store. Special instructions have been added to the NOVA emulator to allow it to read/write into control store, to read/write into the target machine main store, and to "turn on" the real-time processor. In a similar manner, the real-time processor emulation when detecting an error or requiring I/O services can return to the NOVA leaving a flag in the communications buffer to indicate the action required.

Once a real-time program has been debugged, it is often necessary to make a large number of parametric runs to validate or verify the input parameters for various applications of the software. Increased speed may be obtained by deleting capabilities from the emulator and migrating functions from the control program down to the microcoded emulator while at the same time requiring minimal modification to the supporting control program.

MDAC EMULATION LABORATORY QM-1 CONFIGURATION

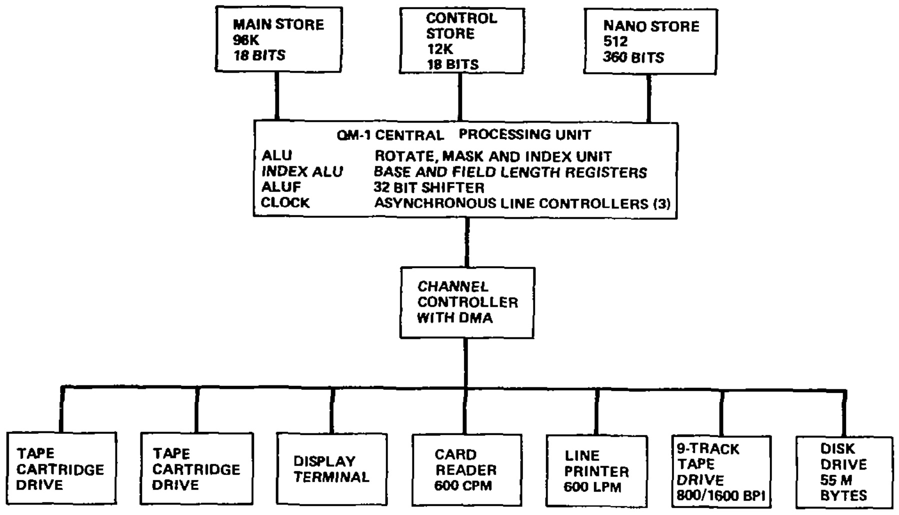


Figure 1

SINGLE USER DUAL EMULATION SYSTEM

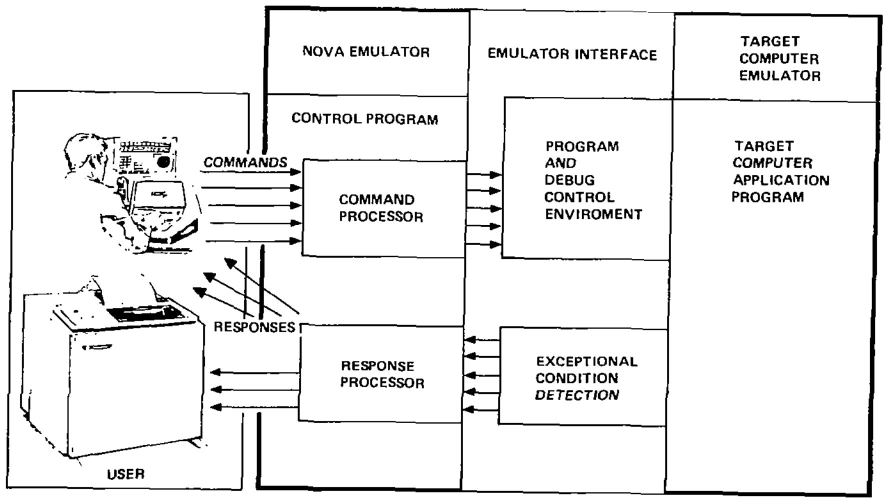


Figure 2

PROGRAM CONTROL FACILITIES

- LOAD DECKS
- CHECKPOINT
- RESTORE
- RESET
- CHANGE PARAMETERS
- EXECUTE
- CONTINUE
- SELECTIVE MEMORY CLEAR
- TELEMETRY STREAM CAPTURE
- DYNAMIC INTERRUPT

Figure 3

DEBUG CONTROL FACILITIES

- BREAKPOINTS
- TRACES (SEQUENTIAL AND BOUNDED)
- SNAP DUMPS
- MEMORY AND REGISTER DUMPS
- MEMORY AND REGISTER EXAMINATION
- MEMORY AND REGISTER MODIFICATION
- MEMORY/FILE COMPARISON
- ERROR CONDITION DETECTION

Figure 4

ERROR CONDITION DETECTION

- MISSING CARDS IN DECK
- ILLEGAL OPERATION
- ILLEGAL INPUT/OUTPUT COMMAND
- ILLEGAL EFFECTIVE ADDRESS
- IMPROPER REGISTER USAGE
- INFINITE LOOPS
- INTERRUPT PROCESSING TOO LONG
- FORCED BREAKPOINT
- HALT INSTRUCTION

Figure 5

SAVINGS VIA EMULATION (DIGS-DELCO PROGRAMS)

COST COMPONENT	SIMULATION ALTERNATIVE	MDAC EMULATION	% SAVINGS
1. DEVELOPMENT TIME - TOOL - COMPUTER PROGRAMS TOTAL	 ——— 4.5 MAN-YR 4.5 MAN-YR (EST)	 1.2 MAN-YR 2.5 MAN-YR 3.7 MAN-YR	18%
2. MACHINE TIME - COMPUTER COST/MIN - RATIO: TOOL TO REAL-TIME - AVERAGE REAL-TIME RUN - NUMBER OF RUNS TOTAL COST	 $7/MIN (S/370) 60:1 1 MINUTE 650 $273K	 $.5/MIN (MDAC QM-1) 4.5:1 1 MINUTE 650 ≈ $1500	OVER 99%
TOTAL COSTS AT $50K/MAN-YEAR	500K	185K	62%

Figure 6

NASA ISIS[*]

W. Joseph Berman
University of Virginia

The National Aeronautics and Space Administration Interactive Software Invocation System (NASA ISIS) is being developed as the central resource of the Multipurpose User-oriented Software Technology (MUST) project. The goal of MUST is to assist in the development of flight software by providing a comprehensive collection of software tools such as requirement analyzers, simulators, static code analyzers, compilers, assemblers, dynamic code analyzers, test case generators, flow charters and report generators. The function of NASA ISIS is to facilitate the use of these tools and to guide the programmer/engineer through an orderly development of flight software.

The major components of NASA ISIS were motivated by considering the flight software development process in detail. This analysis showed that the most important function which the system must perform is to manage the large number of data files (e.g., source modules, object modules, test data and device characteristics) used during software development. In NASA ISIS, these data files are organized into an indexed, hierarchical file system. This file system also allows the storing of multiple "versions" of any file, facilitating both experimental development of a file and retention of a file's history.

The data files which are used in developing flight software can be divided into three basic classes. Source modules and documentation are common examples of textual data. In NASA ISIS, textual files are structured as numbered lines and a text editor allows processing by both linenumber and content criteria. It was found that numeric data files are almost always arranged in a tabular format and that the processing requirements are typically quite straightforward. This suggests that a full database management system is probably unnecessary and, therefore, NASA ISIS provides only a simple and elegant retrieval capability for tabular files. Finally, many data files do not require interactive processing (e.g., object modules).

Once the user has stored the necessary data for a given package, it is often necessary to transform this data into a format acceptable to the tool. While this capability is functionally no different than the interactive editing facilities, it implies the further capability of being able to store complex editing sequences for use by those uninterested in the details of the transformation. In NASA ISIS, these transformations are termed "invocations".

Despite the apparent diversity of the file management, text editing, data manipulation and invocation components of NASA ISIS, there are many primitive functions in common. To take advantage of this overlap and to present the user with a cohesive interface, NASA ISIS was designed as a single,

[*]Research supported by NASA Contract NAS1-14862.

unified system controlled by an "interactive programming language". This language is based upon many of the precepts of PASCAL, but has been especially tailored to the interactive environment and to the special capabilities of the rest of the system. Since this language allows the use of numeric and string literals, types and type constructors, variables, and terminal communications, it is significantly more powerful than most desktop calculators. In addition, the system actually compiles the language into an intermediate code for execution by a virtual machine. This allows both immediate execution of input lines ("command mode") and the storing of the intermediate code for later execution ("compiled mode").

In addition to suggesting the major components of the system, the study of the flight software development process made it clear that any implementation of NASA ISIS needed to have certain characteristics. Since there are several groups within NASA which produce flight software and these groups use different computing systems, it is very important that the system be as transportable as possible. To achieve transportability, the current version of NASA ISIS has been written almost exclusively in a carefully chosen subset of PASCAL. In addition, the implementation takes as little advantage of the host operating system as possible and the interface to the operating system is isolated to a single assembly-language module.

Another aspect of the implementation of NASA ISIS is that it should be as adaptable as possible. As an interactive system is implemented and made available to its user community, there is inevitably feedback from the users on improvements to the system. In the current version of NASA ISIS, the separation of the parsing and execution phases considerably simplifies changes in syntax, and the high degree of modularity (both at the functional and coding levels) allows for significant alterations with only localized impact upon the system.

The current version of NASA ISIS is only an engineering prototype. By first implementing a prototype, it is possible to test many ideas and to determine the system's utility before committing significant resources to the development of a production model. As of November 1978, the engineering prototype is substantially complete and is currently being evaluated at NASA Langley Research Center. By February 1979 it is expected that this prototype will be completed and distributed to the MUST user community for further testing and evaluation. Finally, during the spring of 1979 the system will be re-hosted from its current CDC implementation to an IBM 370/158 to determine the system's transportability.

NASA ISIS

IMPLEMENTATION

- ENGINEERING PROTOTYPE

  TO TEST IMPLEMENTATION TECHNIQUES

  TO ALLOW USER FEEDBACK FOR DESIGN OF PRODUCTION SYSTEM

- ADAPTABILITY

  BY SEPARATION OF PARSING AND EXECUTION PHASES

  BY MODULARITY OF DESIGN AND OF CODING

- TRANSPORTABILITY

  BY USING A SUBSET OF PASCAL AS IMPLEMENTATION LANGUAGE

  BY USING HOST OPERATING SYSTEM AS LITTLE AS POSSIBLE

  BY ISOLATING HOST OPERATING SYSTEM INTERFACE TO SINGLE ASSEMBLY-LANGUAGE MODULE

- STATUS

NOVEMBER 1978:	BEGINNING EVALUATION AT LARC
FEBRUARY 1979:	DISTRIBUTION TO MUST USERS FOR EVALUATION
APRIL 1979:	TEST TRANSPORTABILITY TO IBM 370/158

Figure 1

NASA ISIS

INTERACTIVE PROGRAMMING LANGUAGE

INITIAL DESIGN: INTERPRETED PASCAL

MODIFICATIONS:

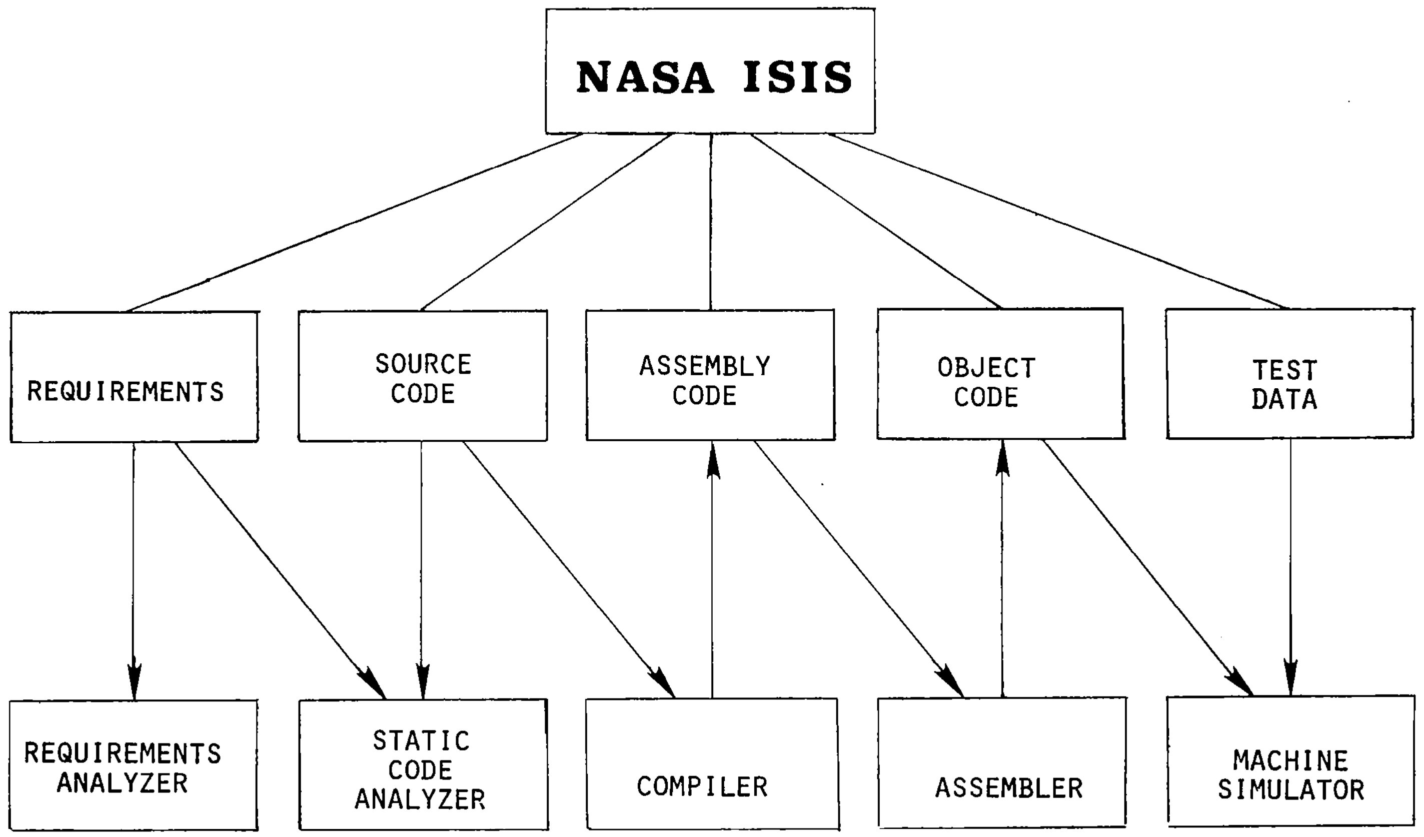
DUE TO INTERACTIVE ENVIRONMENT

- USER PROMPTED FOR DECLARATION OF UNKNOWN IDENTIFIERS
- USER MUST EXPLICITLY ERASE UNWANTED IDENTIFIERS
- BUILT-IN TYPE OF STRING
- MULTI-STATEMENT CONTROL STRUCTURES (LIKE REPEAT-UNTIL)
- PROCEDURES/FUNCTIONS ARE PREPARED AS TEXT, THEN COMPILED

DUE TO OTHER NASA-ISIS CAPABILITIES

- SPECIAL STATEMENTS TO CONTROL FILE MANAGEMENT, DATA EDITING AND JOB SUBMITTAL

Figure 2



A TYPICAL MUST CONFIGURATION

Figure 3

SOFTWARE SYSTEMS DEVELOPMENT AT GRUMMAN AEROSPACE CORPORATION

JACK ROSENBAUM
GRUMMAN AEROSPACE CORPORATION

Embedded Software Systems, including on-board avionics systems, trainers, simulators, and automated test equipment, are a major product line at Grumman Aerospace Corporation. In recognition of the marked increase in software intensity on these projects, a new department was formed to focus attention on, and develop a unified corporate approach to embedded software development. As indicated in Figure 1, the Software Systems Department (SSD) interfaces directly with Engineering and Logistic Support groups to establish Functional Requirements and the necessary hardware/software trades and then designs and develops the appropriate software for integration into the total system.

In accordance with its charter, SSD is now creating a software development environment, outlined in Figure 2, that spans the entire process from initial functional requirements through installation and life cycle maintenance. Figure 3 is an overview of generic embedded systems life cycle and associated documentation milestones. The actual chart is detailed to show each major step, identifiable task, and contractually required reviews and documents, as specified in Navy and Air Force Mil-Spec documents. The chartis the basis for the SSD Manual and Life Cycle approach, and is reflected in departmental policies, guidelines, and management plans. The key elements of our approach to developing software are outlined in Figure 4. The current procedures are clearly in the direction indicated in DOD Directive 5000.29.

Another major activity includes the development of computer-aided system for software development, outlined in Figure 5, that supports our planned approach. The automated environment currently includes a series of requirements development, design development, configuration management, and documentation tools on a large central host (IBM/Amdahl) installation that allows interactive use a project sites. Program development and testing for Trainers is generally performed on-site in dedicated Software Development Facilities incorporating the same hardware as the target configuration. A development facility utilizing FASP on a CDC computer is also available. Programming tools that support the development process are designed to interface with, and provide status and "as built" data to the IBM/Amdahl for project and configuration management.

The key software tools currently installed on the system, and their application, are described in Figure 6. Our basic philosophy is to procure available systems wherever possible, and integrate them into our overall system. Internal development is undertaken to provide interfaces or to satisfy specific needs normally not addressed in currently marketed systems. Wherever possible, tools developed on projects are generalized and included in our facility. An ongoing technology project has as its major objective the long range development of a generic environment.

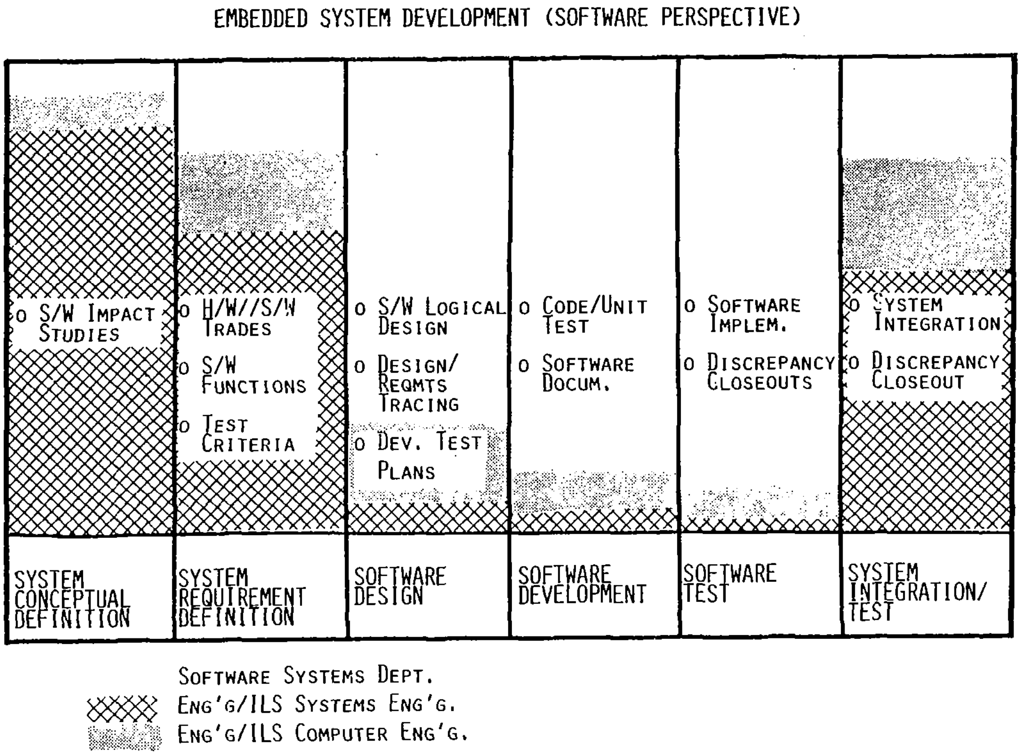


Figure 1

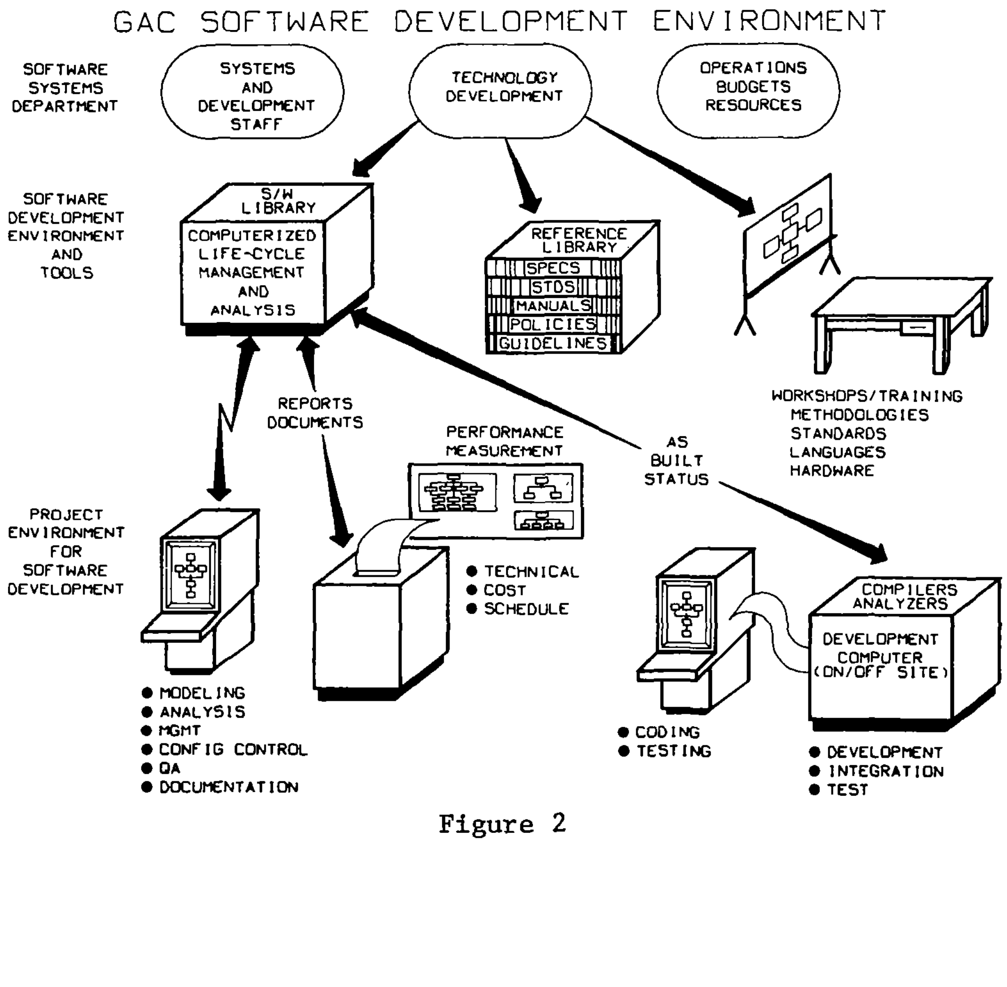


Figure 2

SOFTWARE SYSTEM LIFE CYCLE OBJECTIVES

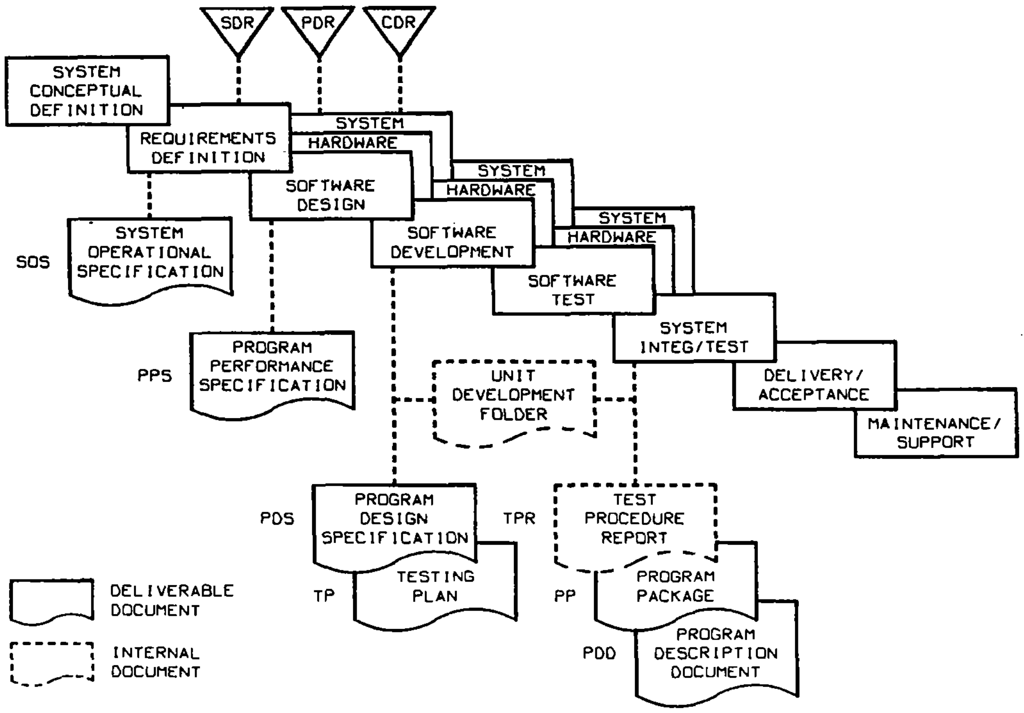


Figure 3

KEY LIFE CYCLE STANDARDS

PROGRAM PERFORMANCE SPECIFICATION
- HARDWARE/SOFTWARE TRADES AND FUNCTION SELECTION
- FUNCTIONAL DESCRIPTIONS (HIERARCHY, PROCESS FLOW)
- PERFORMANCE CRITERIA

PROGRAM DESIGN SPECIFICATION
- LOGICAL DECOMPOSITION TO UNIT MODULE LEVEL

DEVELOPMENT/TEST PLANS
- MODULES LOGICALLY GROUPED FOR TESTING. (FUNCTIONAL THREADS)

DEVELOPMENT/SOFTWARE INTEGRATION TEST
- USE OF DEVELOPMENT FACILITY AND SITS
- DEVELOPMENT FOLDER/AUTOMATED TRACKING
- FORMAL QA AND BONDING

TRACKING
- KEYED TO PERFORMANCE ON DEVELOPMENT UNITS
- TIMING, LINES OF CODE, CORE ESTIMATES

Figure 4

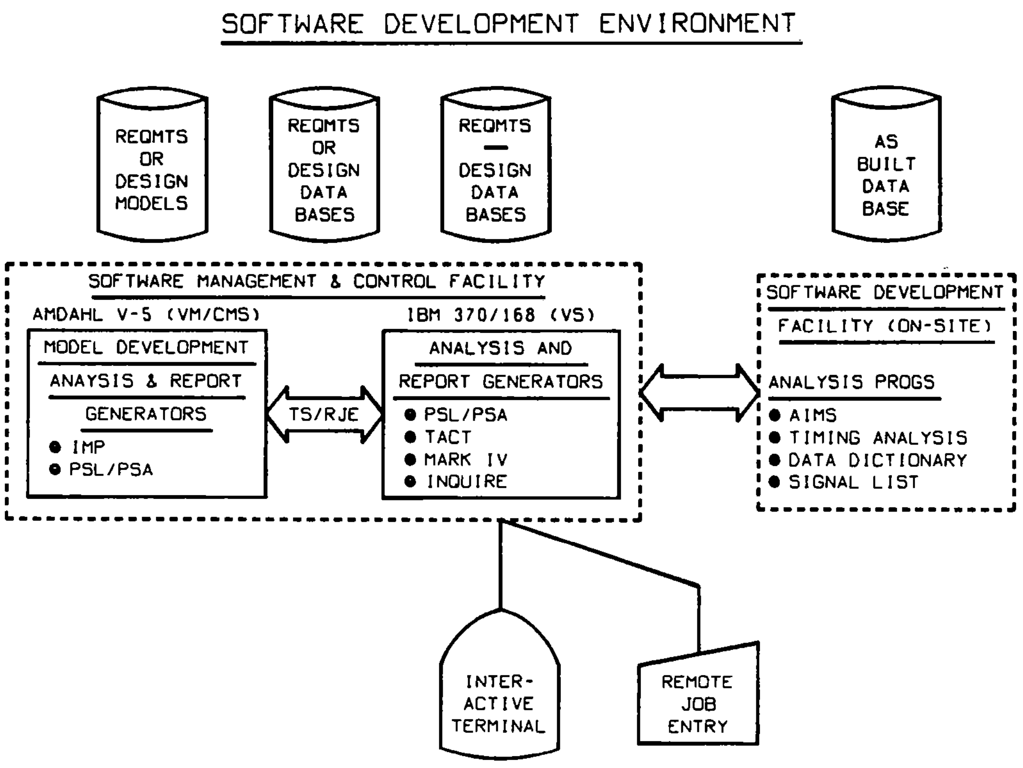

SOFTWARE DEVELOPMENT ENVIRONMENT
REQMTS OR DESIGN MODELS
REQMTS OR DESIGN DATA BASES
REQMTS — DESIGN DATA BASES
AS BUILT DATA BASE
SOFTWARE MANAGEMENT & CONTROL FACILITY
AMDAHL V-5 (VM/CMS)
IBM 370/168 (VS)
MODEL DEVELOPMENT ANAYSIS & REPORT GENERATORS
● IMP
● PSL/PSA
TS/RJE
ANALYSIS AND REPORT GENERATORS
● PSL/PSA
● TACT
● MARK IV
● INQUIRE
SOFTWARE DEVELOPMENT FACILITY (ON-SITE)
ANALYSIS PROGS
● AIMS
● TIMING ANALYSIS
● DATA DICTIONARY
● SIGNAL LIST
INTER-ACTIVE TERMINAL
REMOTE JOB ENTRY

Figure 5

KEY SYSTEMS RELATED SOFTWARE TOOLS

0 IMP

INTERACTIVE MODELLING PROGRAM (IMP) DEVELOPS HIEARCHIAL MODELS/CONTROL MAPS TRANSFORMS TO PSL OR TACT

0 PSL/PSA

PROBLEM STATEMENT LANGUAGE/ANALYSIS USED FOR DOCUMENTING AND ANALYZING FUNCTIONAL REQUIREMENTS AND TRACING TO DESIGN

0 TACT

TRACKS AND STATUSES MODULES OF HIERARCHIAL STRUCTURE THROUGH CODE, UNIT TEST AND INTEGRATION.

0 AIMS

ANALYZES CODED MODULES FOR VARIABLE USAGE (COMMON, LOCAL, I/O, ETC.) FOR USE IN CONFIGURATION CHECKING

0 SIGNAL AND MNEMONIC DICTIONARIES

DATA DICTIONARIES FOR AS BUILT CONFIGURATION

Figure 6

LANGUAGE TOOLS

WHERE THE LEVERAGE IS

Fred H. Martin
Intermetrics, Inc.

SUMMARY

This paper contains an overview of the "Tools and Language Processors" Session of this workshop.

DISCUSSION

The life cycle development and utilization of embedded software can be viewed as a succession of phases with appropriate feedback among the phases. Roughly speaking, these can be identified as requirements, specifications, design, code and test, validation and maintenance. It is now generally acknowledged that at least 50% of the life cycle cost must be allocated to activities following initial release of the program, viz., for validation and maintenance. This would include listings and support documentation for maintenance, independent verification and validation, operational support, program enhancements and general maintenance. Where the life time of the software is expected to be very long, e.g., 20 year military systems, the figure 50% is probably conservative.

All of the software phases have been the subject of intense study over the past few years. Significant results are usually manifested by the appearance of a "software tool" to aid in accomplishing a particular phase. Indeed, these tools are the subject of this workshop.

It is interesting to note the potential cost payoff within each phase, i.e., what contributions can reasonably be expected with respect to lowering overall software costs. (Improving reliability is viewed as an important factor in lowering cost.) It is in this context that the leverage of a tool, or the ratio of eventual benefit to initial cost must be kept in mind.

Intuitively, the further back in the chain of software development one can introduce an improvement, the greater will its cumulative effect be in improving the process, thereby reducing the cost. Obviously, a coding error caught by a compiler will necessitate the cost of a recompile, probably orders of magnitude less, however, than the cost associated with finding the hidden bug during validation, or worse -- operations. Receding further back from the coding process into design, specifications or requirements should produce correspondingly larger savings.

But what of the practicability of such improvements and the necessary cost to achieve them? Without a doubt, the coding process is the most mature and the most amenable to immediate results. A veritable revolution has occurred in the last 10 years with the recognition of the value of structured programming and design. The readily available tool has been -- language. Capable higher-order languages with strict data typing, finite control structure and inherent redundancy have both improved programmer efficiency and eliminated whole classes of errors. The elimination of a family of errors can reduce dramatically the scope of necessary validation while at the same time improve, by definition, software reliability.

Unfortunately, the connections between requirements and specifications to language, on the one hand, and language to validation, on the other, are still tenuous today. No satisfactory expression of requirements has yet emerged, and although many promising methodologies abound for specifications and design, the translation to higher-order language implementation (or execution) is still a human task. With respect to verification, deriving the domain of necessary tests for a non-trivial program which spans both operational requirements and potential anomalies is still an unsolved problem. Usually, it's simply a question of budgets. Validation must be a finite activity, and a large program can be validated to the tune of 5, 50, or 500 man years.

While I'm not at all discouraged at the work being done in requirements, specifications and validation, and the excellent tools being developed and proposed, I believe that is it here -- in languages -- where the leverage is, i.e., the greatest payoff for the least investment. The reasons are simple; the problems are easier, the field is more mature and we are dealing with the final product itself, not what it could or should be, or how it's supposed to behave when stimulated, like a black box.

CONTRIBUTED PAPERS

The papers in this session reflect very practical attempts to capitalize on the advantageous position that language has in the life cycle process.

The first on "path expressions" recognizes that the source programming language is the best place to express the complex interactions necessary in real-time programming. Long ignored and relegated to the assembly language executive or operating system, several languages have attempted quite successfully (HAL/S, PEARL, CONCURRENT PASCAL) to bring the light of day to real-time control. The DoD-1 designs are now struggling with this important feature. Path expressions allow synchronization and interaction of shared resources to be expressed right within the static structure of a program and thereby in an apparent and understandable form. The dynamics of asynchronous operations need not be first discovered when the program executes.

Our second paper, again tries to place more information, and increased redundancy in the programming language, so as to catch automatically a larger class of errors and improve diagnostics. Attention is paid to reformatting the listing, providing hierarchical reports and to the preparation of tables for program analysis. Note the immediate results to be gained through the language and compiler. All the derived data concerns the actual product. Much of the analysis can be carried out automatically; it will reflect the current software immediately, and it is accurate. It is gratifying to me to see these features being added to PASCAL with the promise of more. One of the objectives in the HAL/S design was to "wring out" of the program source every conceivable aid toward understanding and verifying the flight software.

The next paper is somewhat of a change of pace but actually within the theme. Most flight computers require assemblers and linkage editors for coping with the output of a compiler or for handling necessary assembler language code. Unfortunately, because most machines differ in detail, assemblers are written anew. The enhanced META Assembler described in this paper employs some of the powerful "front-end" techniques found in compilers and allows full user flexibility over variable form and content. This should reduce materially the cost and schedule for development of arbitrary assemblers. I understand the META Assembler is readily available and its rehostability and retargetability have been demonstrated. In a sense it is a basic tool, often overlooked in the context of language.

The fourth paper, in a way, is a synthesis of many of the points discussed here. The Universal Flowcharter is itself designed and built using the HOS methodology and the AXES specification language. Specifications have been implemented, for now, by hand in PASCAL. The result is a universal documentation aid which can flow chart any block-oriented higher order language. The charts should prove useful during design, verification, and maintenance. Once again, the power and versatility of language is evident. A formal grammar, expressed semantic rules, and inherent redundancy provide enough information for the automatic graphic algorithms and descriptive "concordances" presented in the paper.

The final paper reminds us all that while programming languages offer great potential, it's the compilers that deliver reality, and compiler production has been very expensive, at best. In today's technology the compiler "front end" from source to immediate language (IL) is well understood and most modern compilers employ this machine independent "Phase 1". However, the IL must then be translated to a variety of target machine instruction sets through a code generating process. Since each target is usually quite different, much of the code generator must be built from scratch.

This paper describes a method of designing a code generator using the latest software methodologies so that classes of machines can be paramaterized. This is accomplished by "hiding" the specific details of machine architecture thereby rendering a substantial portion of the code generator target machine independent.

The research described first translates the HAL/S IL (HALMAT), as an example, into a more amenable <u>second</u> IL for a particular class of machines. Thereafter, new code generators for that class can be implemented more economically. The implication is that a finite set of "second IL's" should suffice to cover most of the common machines.

Work is continuing in developing a tool to generate automatically good machine code from a defined IL.

CONCLUDING REMARKS

As chairman, I wish to express my appreciation to the contributors for their interesting papers. I look forward to the ensuing discussions at the Workshop.

PATH EXPRESSIONS FOR REAL-TIME PROGRAMMING

R. H. Campbell
University of Illinois at Urbana-Champaign

Many aerospace embedded computer systems involve the design and construction of system software requiring asynchronous processes, synchronization, co-ordination and communication between processes, and guaranteed performance within a set of real-time constraints. Such design is difficult, expensive and error prone.

This paper presents research into mechanisms which aid the description and analysis of real-time aerospace software and which can be adapted to provide appropriate extensions to current programming languages. These mechanisms are based upon path expressions. An experimental language called PATH PASCAL was developed as a testbed to study the implementation and interactions of these mechanisms and to observe their use in a realistic environment. PATH PASCAL extends PASCAL to include concurrent processes, path expressions to provide synchronization and co-ordination, and an encapsulation mechanism which, together with the path expressions, provides synchronized access to shared, protected data. PASCAL was chosen because, in our opinion, it is a well-engineered compromise between expressiveness, generality, efficiency, clarity, and simplicity.

To gain practical experience with these mechanisms, we are conducting two experiments. Several real-time systems programs are being written for a PDP-11, including I/O device drivers, schedulers, and network communications. The execution times of these programs will be estimated and measured to provide statistics for research into deadlines. In a separate project, the language will be used by system programming students. Their experiences will provide valuable insights into the appropriateness of these mechanisms.

The PATH PASCAL testbed has been successfully implemented together with both a simulated real-time environment and asynchronous I/O. The compiler translates PATH PASCAL to an intermediate code which can be interpreted on a CYBER 175, or assembled into machine code for execution on a PDP-11. The feasibility of using Open Paths to specify synchronization has been shown by our implementation. A practical evaluation of this mechanism can now be undertaken and comparisons made with other synchronization techniques. The implementation and language can also be used to provide a basis for future experiments involving modifications to the synchronization mechanism and further language extensions.

Future experiments will include more sophisticated Path expressions, a guaranteed deadline mechanism which provides a graceful degradation of service while maintaining the guarantee of meeting deadlines, error recovery mechanisms based on recovery blocks, language mechanisms to allow device I/O routines to be written in a high-level language and possible generalization of the mechanisms to include networks of systems.

To conclude, PATH PASCAL is a feasible testbed for investigating mechanisms to synchronize concurrent processes in real-time aerospace systems. The language mechanisms we have implemented interact cleanly with the PATH PASCAL language and lead to a straightforward implementation. The PDP-11 PATH PASCAL runtime system is only slightly different from that of ordinary PASCAL. PATH PASCAL is portable and can be used on a variety of computers. Finally, this experience indicates that similar modifications can be made to existing block-structured avionic languages.

PURPOSE:

*Develop Language Mechanisms for Aerospace Systems.

*Synchronize Co-operating Concurrent Processes.

Separate Synchronization Specification -------	Open Path.
Shared Data Structure and Encapsulation ------	Object.
Concurrent Execution ------------------------	Process.

*Real-Time Constraints.

Figure 1

MOTIVATION:

*Increasing Complexity (Concurrency, Real-Time Constraints) causes

Costly, Difficult and Error Prone Software Development.

*Realistic Testbed for Experimentation:

Observe Use of Language Mechanism.

Pascal:	Real Language.
	Small, Modifiable.
	Well-Engineered.

Figure 2

FUTURE RESEARCH:

*Alternate Path Notations.

*Guaranteed Deadline Mechanism.

*Error Recovery Mechanism.

*Mechanism for Programming Device I/O Routines.

*Networks.

Figure 3

PROGRESS AND EVALUATION:

*Path Pascal Language with Simulated Real-Time Environment and Simulated Asynchronous I/O.

*CYBER 175: Interpreted Intermediate Code.

*PDP-11: Executed Machine Code.

*Open Paths are Feasible.

*Practical Evaluation and Comparison is Possible.

*Basis for Future Experiments.

Figure 4

FUTURE APPLICATIONS:

*Real-Time Experimentation on PDP-11.

I/O Device Drivers.

Schedulars.

Communications.

Estimating and Measuring Execution Times.

Deadlines.

*Systems Programming Class Projects.

Figure 5

CONCLUSIONS:

*Path Pascal Testbed Approach is Feasible.

*Extensions to Pascal:

Clean Interaction.

Straightforward Implementation.

Small Additions to Run-Time System required for PDP-11.

*Mechanisms Suitable for Block-Structured Languages.

Figure 6

VERIFIABLE PASCAL

Sabina H. Saib
General Research Corporation

Verifiable PASCAL contains enhancements to the programming language PASCAL designed to allow for more extensive error checking than is possible in PASCAL. The language is implemented in a processor that generates PASCAL.

The extensions include:

- improved structures which are easier to write and result in more readable code
- executable assertions which can be used to report exceptions during testing and also can be used by a program verifier
- data access restrictions which limit the access rights and operations on data
- units qualifiers which declare the physical units of variables thereby making units consistency checking possible.

VPASCAL source programs may be executed identically to PASCAL source programs after they have been processed by the VPASCAL preprocessor. Functions of the VPASCAL preprocessor include translation, generation of an enhanced source-code listing and static module hierarchy report, interface to the Software Quality Laboratory (SQLAB) verification tools, and reformatting of VPASCAL source code for program maintenance. The services implemented include:

1. Translation of all VPASCAL control statements into standard PASCAL while passing other PASCAL statements ummodified.
2. Indented and annotated listing of the VPASCAL source code.
3. Module hierarchy report and directory showing the static nesting structure of modules and a directory to the indented listing for module sections (i.e., heading, LABEL, CONST, TYPE, VAR, and statement).
4. Assertion statements translated into executable form selectively; the default is to change assertions into PASCAL comments.
5. Interface file prepared for input to other SQLAB tools, thereby permitting program verification analysis.
6. Reformatted source code to improve legibility in program maintenance.

IMPLEMENTATION OF VERIFIABLE PASCAL PREPROCESSOR

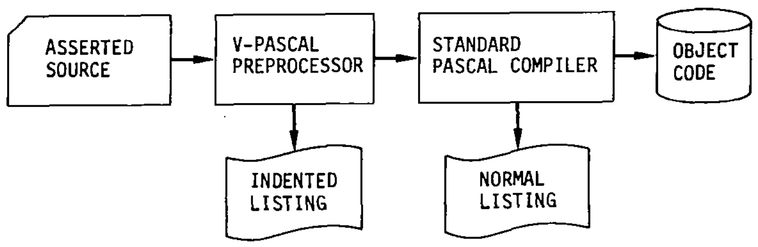


Figure 1

VERIFIABLE PASCAL PROCESSOR CAPABILITIES

- TRANSLATES VERIFIABLE PASCAL TO STANDARD PASCAL WITH OPTIONAL EXPANSION FOR EXECUTABLE ASSERTIONS
- TRANSLATES STANDARD PASCAL TO VERIFIABLE PASCAL
- GENERATES INTERFACE FILE (FOR OTHER SQL TOOLS) WITH ANALYZED TEXT AND SYMBOL DESCRIPTIONS
- GENERATES FORMATTED LISTING, TEXT DIRECTORY AND MODULE STRUCTURE, LANGUAGE USAGE
- LARGE APPLICATIONS INCLUDE
  - PASCAL COMPILER - 10000 LINES - 139 PROCEDURES - 8 LEVELS
  - VERIFIABLE PASCAL PROCESSOR - 6000 LINES - 170 PROCEDURES - 3 LEVELS
- VERY EASY TO ADAPT FOR LANGUAGE CHANGES
- EASY TO ADAPT FOR CHANGES IN PROCESSING REQUIREMENTS

Figure 2

HOW VERIFIABLE PASCAL DIFFERS FROM STANDARD PASCAL

- CONTROL STRUCTURES WITH UNIQUE TERMINATORS
  (IF...END IF, FOR...END FOR)
- CONTROL STRUCTURE EXTENSION
  (IF...ORIF...ELSE...END IF)
- EXECUTABLE ASSERTIONS
  (ASSERT, INITIAL, FINAL)
- DATA ACCESS CONSTRAINTS
  (INPUT, OUTPUT)
- PHYSICAL UNITS SPECIFICATION
  (UNITS)
- TRANSLATION OPTIONS

Figure 3

WHY VERIFIABLE PASCAL

PASCAL ALREADY HAS FEATURES AMENABLE TO VERIFICATION

TYPE CHECKING
CLEAN SYNTAX
CALL CHECKING

PASCAL HAS USEFUL DATA STRUCTURES

RECORD
POINTER

PASCAL COMPILERS ARE BECOMING COMMONLY AVAILABLE

PASCAL IS MISSING SOME FEATURES

AUTOMATICALLY INDENTED LISTINGS
DATA ACCESS RIGHTS
UNITS SPECIFICATIONS
LOGICAL ASSERTIONS

PASCAL ALSO HAS A FEW SYNTAX PROBLEMS

AMBIGUOUS IF STATEMENT
INTEGERS USED FOR BOTH STATEMENT LABELS AND CASE SELECTION

Figure 4

VERIFIABLE PASCAL EXAMPLE

```
BEGIN (* COUNT COLORS *)
     INITIALIZE ; (* LOOp TO END OF FILE *)
     READALPHA ( NAME ) ;
     WRITELN ( "1  NAME      HAIR" ) ;
     WHILE NOT EOF DO
     . READALPHA ( HAIR ) ;
     . WRITELN ( "  " , NAME , HAIR ) ;
     . NUMBERREADS := NUMBERREADS + 1 ;
     . ANS := FALSE ;
     . FOR COLOR := BLACK TO WHITE DO
     . . IF HAIR = COLUMNS [ COLOR ] THEN
     . . . COUNT [ COLOR ] := COUNT [ COLOR ] + 1 ;
     . . . ASSERT ( COUNT [ COLOR ] > 0 ) AND ( COUNT [ COLOR ] <= NUMBERREAD ) ;
     . . . ANS := TRUE
     . . ENDIF
     . END FOR ;
     . ASSERT ANS
     . . FAIL
     . . . WRITELN ( " CARD IN ERROR " ; NAME , " " , HAIR ) ;
     . . END FAIL ;
     . READALPHA ( NAME ) ;
     END WHILE ; (* PRINT RESULTS *)
     WRITELN ;
     FOR COLOR := BLACK TO WHITE DO
     . WRITELN ( " FOR COLOR " , COLUMNS [ COLOR ] , "COUNT IS" , COUNT [ COLOR ] ) ;
     END FOR ;
     OUTPUTS COUNT ;
END ; (* COUNT COLORS *)
```

Figure 5

GRAMMAR FOR VERIFIABLE PASCAL IF STATEMENT

```
<IFSTATEMENT> =
          ≡IF≡ <PREDICATE> ≡THEN≡ <STATEMENTLIST>
          *[ ≡ORIF≡ <PREDICATE> ≡THEN≡ <STATEMENTLIST> ]*
          [ [ ≡ELSE≡ <STATEMENTLIST> ] v VIDE ]
          ≡ENDIF≡
```

Figure 6

A NEW META ASSEMBLER

K. V. Smith, Z. Jelinski, J. B. Churchwell, and S. Park
McDonnell Douglas Astronautics Company-West

This paper describes major improvements to the existing Meta Assembler in order to provide a generalized system for the assembly of computer programs for target machines. The system is host portable and target reconfigurable. The improvements provide a user-oriented syntax definition capability. This is accomplished by adding an assembly language translator which allows a user to describe any assembly language syntax in a meta language. A generalized linkage editor is developed to provide a flexible tool for the user to link assembled programs and make changes, modifications and corrections to individual modules without the necessity of reassembly of the whole program each time.

The concept of the Meta Assembler originated at NASA MSFC some six years ago and over the last several years the system was developed and applied to a number of hosts and target machines. The idea was to have a software development tool resident on a host machine to prepare programs for a number of target machines thus saving assembler development costs for each of the targets.

In spite of its usefulness, the system had some serious shortcomings namely the Meta Assembler used a language independent syntax for directives (pseudo ops), macros and labels because these features could differ greatly from one assembly language to another. For this reason, existing assembly language programs had to either have the source for these differences rewritten or a syntax preprocessor had to be written to change them.

NASA has therefore sponsored a major enhancement to the Meta Assembler. The new Meta Assembler will now include a user oriented syntax definition capability. This state of the art technique includes the assembly language definition using a meta language. A statement in the meta language may define user types, parameters, table entries, target machine characteristics, assembler language symbols, semantic functions and comments. The meta language definition is processed by the meta language processor, ALLDEF, building a "dictionary" which provides the basis for the assembly process. A generalized parser-ALLTRAN will complete the translation process by performing the alternative first pass of the cross assembly. The output of the generalized parser will be an intermediate language (IL) data set such that the existing second pass of the Meta Assembler can complete the cross assembly by converting the IL into the object data file and generate a program listing as shown in Figure 1. A second enhancement task is to construct a Generalized Linkage Editor which is a multi-function utility designed to aid the Meta Assembler user in the creation and maintenance of software systems built from Meta Assembler formatted object modules. The result of this effort will be a single Meta Assembler program and a Linkage Editor program which operate in the environment of a large scale computer and support software development for flight and ground checkout computers.

META ASSEMBLER CONFIGURATION

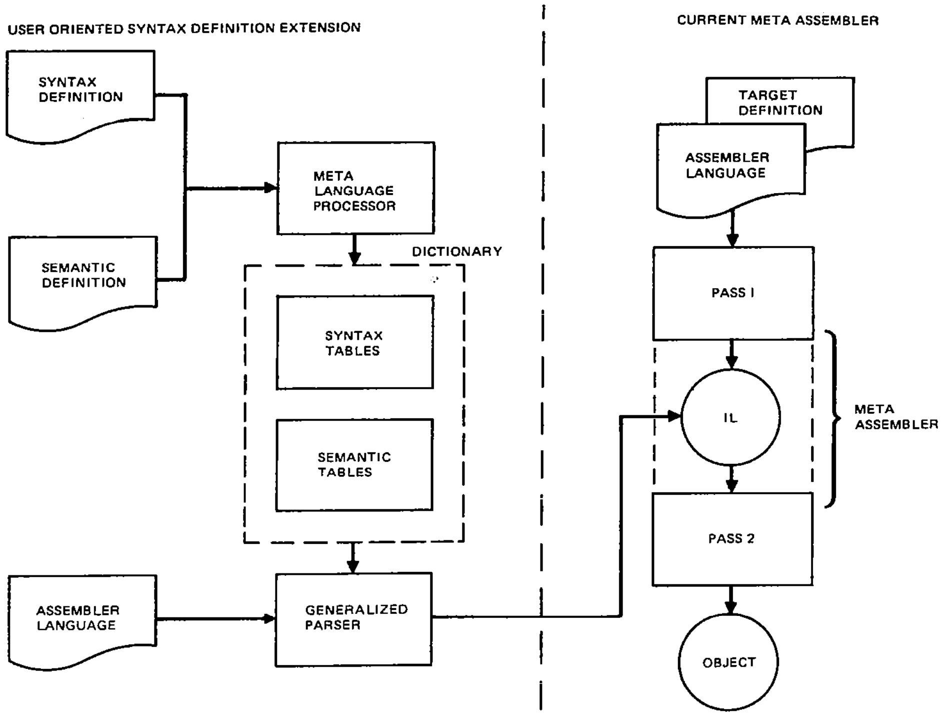


Figure 1

ALLDEF PROCESSOR

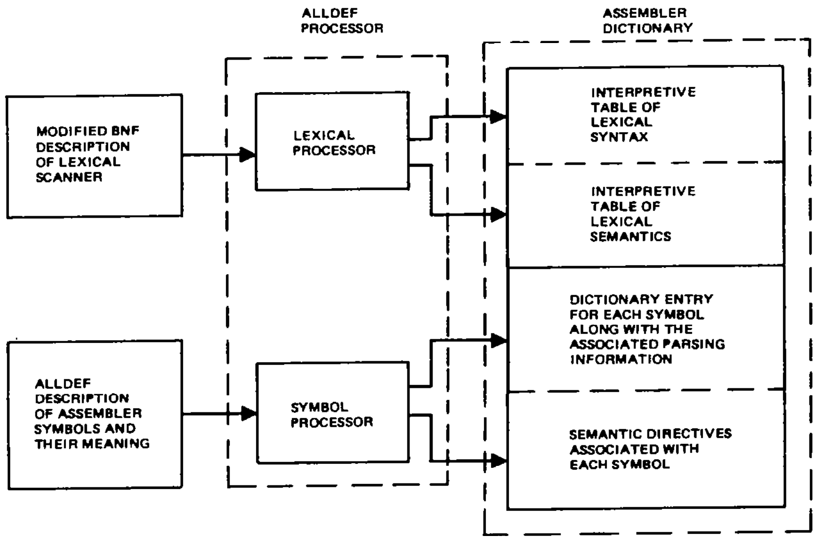

ALLDEF PROCESSOR
ASSEMBLER DICTIONARY
MODIFIED BNF DESCRIPTION OF LEXICAL SCANNER
LEXICAL PROCESSOR
INTERPRETIVE TABLE OF LEXICAL SYNTAX
INTERPRETIVE TABLE OF LEXICAL SEMANTICS
DICTIONARY ENTRY FOR EACH SYMBOL ALONG WITH THE ASSOCIATED PARSING INFORMATION
ALLDEF DESCRIPTION OF ASSEMBLER SYMBOLS AND THEIR MEANING
SYMBOL PROCESSOR
SEMANTIC DIRECTIVES ASSOCIATED WITH EACH SYMBOL

Figure 2

ALLTRAN PROCESSOR

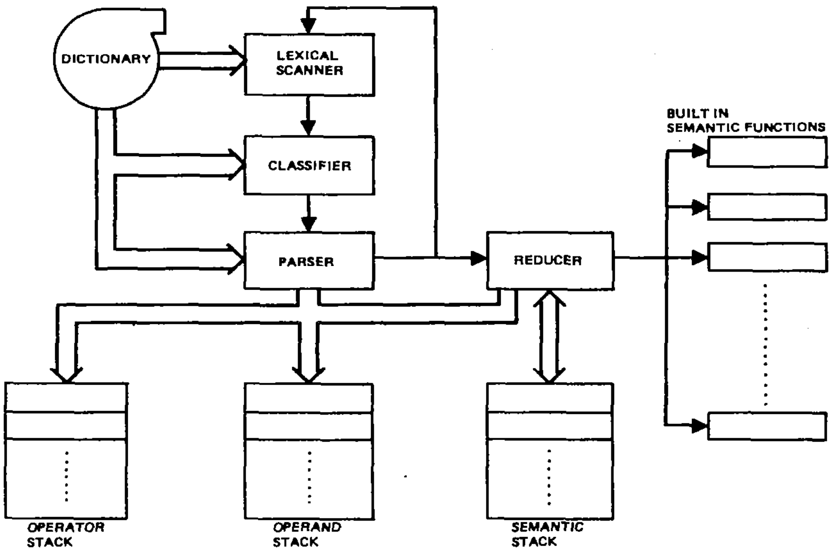


Figure 3

FLOW THROUGH THE GENERALIZED LINKAGE EDITOR

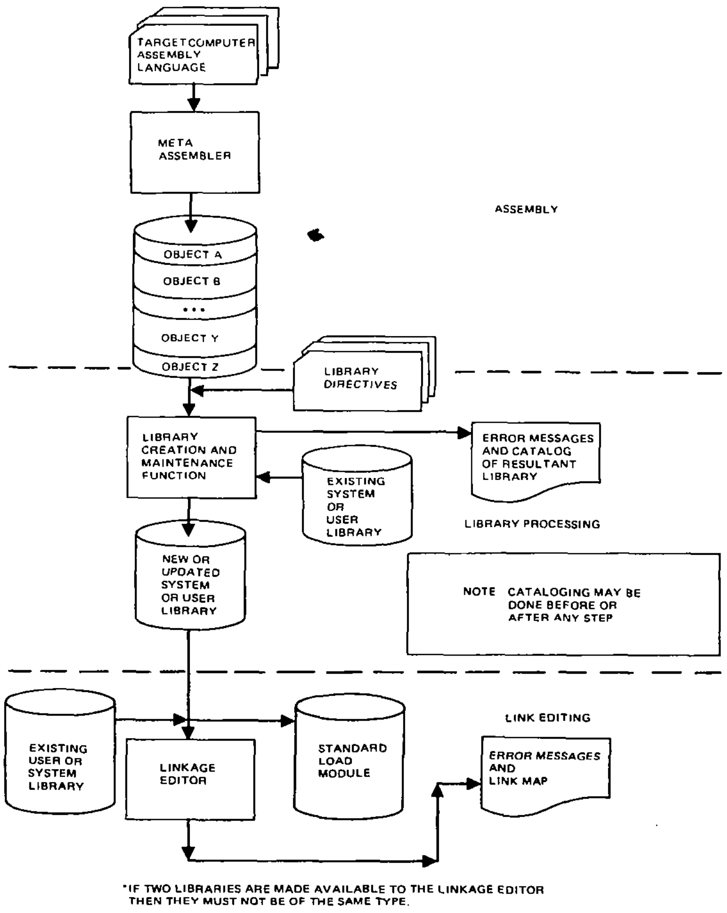

TARGET COMPUTER ASSEMBLY LANGUAGE
META ASSEMBLER
ASSEMBLY
OBJECT A
OBJECT B
...
OBJECT Y
OBJECT Z
LIBRARY DIRECTIVES
LIBRARY CREATION AND MAINTENANCE FUNCTION
ERROR MESSAGES AND CATALOG OF RESULTANT LIBRARY
EXISTING SYSTEM OR USER LIBRARY
LIBRARY PROCESSING
NEW OR UPDATED SYSTEM OR USER LIBRARY
NOTE CATALOGING MAY BE DONE BEFORE OR AFTER ANY STEP
LINK EDITING
EXISTING USER OR SYSTEM LIBRARY
LINKAGE EDITOR
STANDARD LOAD MODULE
ERROR MESSAGES AND LINK MAP
*IF TWO LIBRARIES ARE MADE AVAILABLE TO THE LINKAGE EDITOR THEN THEY MUST NOT BE OF THE SAME TYPE.

Figure 4

A UNIVERSAL FLOWCHARTER

J. Rood, T. To, and D. Harel
Higher Order Software, Inc.

A Universal Flowcharter has been developed for the MUST system [1].* The Flowcharter was specified in [2] using the HOS control-tree specification language (c.f. [3]) and has been implemented in the PASCAL programming language. Given a description of an input programming language grammar, the source code of a program written in that language and a partial semantic description of the language, the Flowcharter acts as a graphic documentation tool for that program. Versions using a HAL/S grammar and a PASCAL grammar are currently running on the MUST system. Two types of output modes are available: line-printer output and CALCOMP plotter output.

Instead of control flow being represented by lines that loop back on themselves at certain points, the output of the Flowcharter has a tree format appropriate for representing the control flow in a structured program.

The algorithm used by the Flowcharter (see [3]) is new in the sense that the syntactic analysis performed makes use of extensive semantic information too. The description of the input programming language grammar is augmented for each production rule in the grammar with a semantic rule which describes the relevant semantic contents of the corresponding syntactic construct. Thus, for example, corresponding to a syntactic rule for the standard IF...THEN...ELSE statement is the semantic information which specifies that this is a conditional statement and is to be plotted accordingly. The Flowcharter performs an LR(1) bottom-up parse on the input program, gathering the semantic information as it goes along. Control-flow information is retained by substituting parts of the input program into standard "templates." The decisions as to exactly which type of template and in which form the substitutions are carried out, depends on the accumulated semantic information. Nested substitutions of the templates result in a tree-representation of the program which is used to drive the plotting (or printing) of the Flowcharter output. Seven types of templates are currently used, corresponding to seven basic control-flow structures: block statements, conditional statements, iterative statements, concurrent statements, non-deterministic choice statements, and procedure declarations [see figures].

Other semantic information, including the names of variables defined locally to a procedure, those assigned in a procedure, those referenced in a procedure, and which procedure calls what other procedures, is collected at each production. At the end of the parsing, these data are collated and a set of "concordances," one for each procedure, is produced. The concordance of a procedure is printed after its control-flow structure is plotted.

The plotting/printing is basically driven by the templates. For each specific type of template, a routine is called to handle the corresponding construction. Nested templates are printed/plotted in indented columns so that the control flow of the program will appear explicitly. In addition, many parameters, such as the number of columns to be printed/plotted on a page, the choice of a long or short concordance, and detail tailoring of printing/plotting formats can be specified by the user.

REFERENCES

1. Straeter, T.; Foudriat, E., and Will, R.: MUST - An Integrated System of Support Tools for Research Flight Software Engineering. A Collection of Technical Papers, AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, Los Angeles, CA, Nov. 1977.

2. Harel, D.; and Pankiewicz, R.: A Universal Flowcharter. TR-11, Higher Order Software, Inc., Nov. 1977.

3. Hamilton, M.; and Zeldin, S.: AXES Syntax Description. TR-4, Higher Order Software, Inc., Dec. 1976.

*Number in brackets indicates reference.

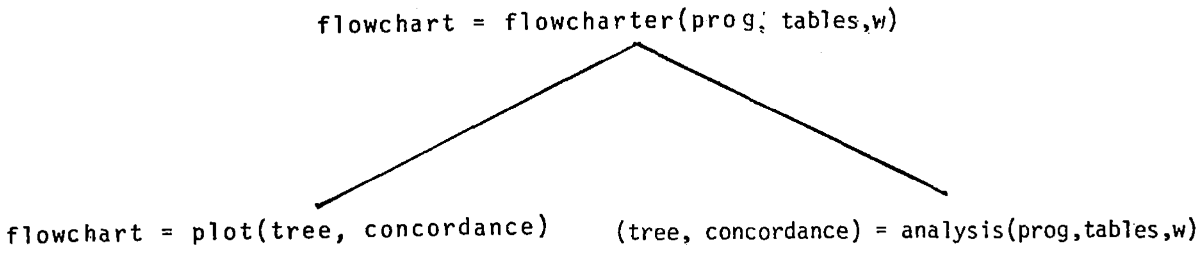

flowchart = flowcharter(prog, tables,w)
flowchart = plot(tree, concordance)
(tree, concordance) = analysis(prog,tables,w)

Figure 1

Though simple to read, structured design diagrams offer the following advantages:

- Clear presentation of the block structure and levels of nesting in a program.
- Full illustration of the decision structure and flow of control in a program.
- Comprehensive tabulation of the scope of variables and functional and data dependencies amonn modules.
- *Automated documentation of programs to promote* standardization and expediate verification.

Figure 2

FLOWCHARTER SPECIFICATION

- HOS methodology
- Specification language (AXES)
- Equivalent control map representation
- Separates specification from implementation
- Algorithm specified was free of serious problems
- Algorithm based on well developed theory of parsing
- Use of embedded metasymbols in program text
- Algorithm specified to receive variable programming language definitions as input

Figure 3

FLOWCHARTER IMPLEMENTATION
1. Analysis

- Flowcharter (FC) parses input text using LR(1) grammar
- Analysis is driven by a set of input tables BNF, "BSF", etc.
- Each production rule has a semantic template associated with it
- Each parsing reduction substitutes text into template
- Final output of analysis is linear program text with embedded metasymbols

FLOWCHARTER IMPLEMENTATION
2. Plot

- FC printed output is naturally represented by a tree
- Plot uses recursive procedures to go from linear representation to tree representation
- Tree representation used to drive printing in real time
- User parameters control output format
- "Concordance" contains symbol reference information

Figure 4

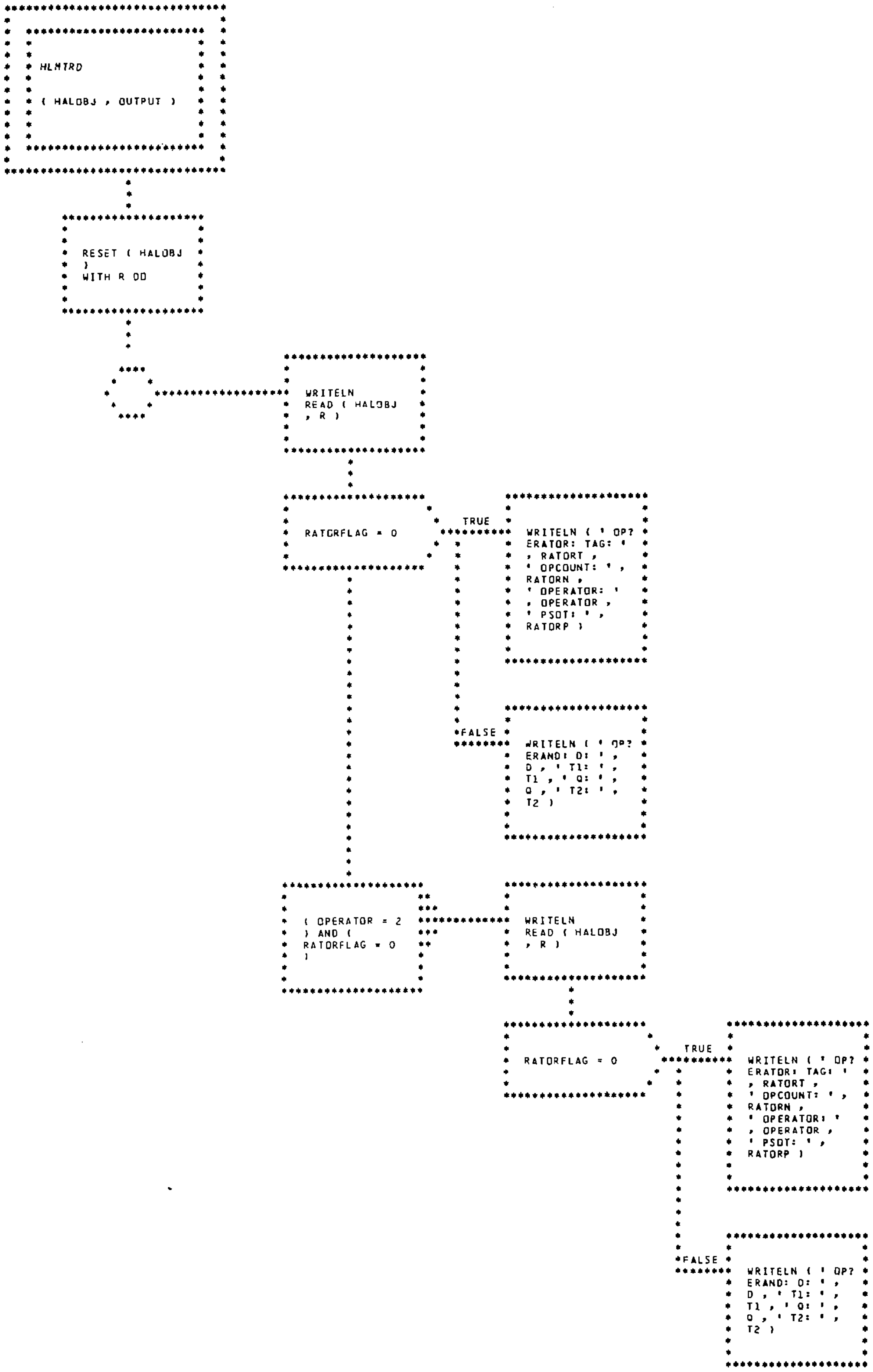

HLMTRD
(HALOBJ , OUTPUT)
RESET (HALOBJ)
WITH R DO
WRITELN
READ (HALOBJ , R)
RATORFLAG = 0
TRUE
WRITELN (' OP? ERATOR: TAG: ' , RATORT , ' OPCOUNT: ' , RATORN , ' OPERATOR: ' , OPERATOR , ' PSOT: ' , RATORP)
FALSE
WRITELN (' OP? ERAND: D: ' , D , ' T1: ' , T1 , ' Q: ' , Q , ' T2: ' , T2)
(OPERATOR = 2) AND (RATORFLAG = 0)
WRITELN
READ (HALOBJ , R)
RATORFLAG = 0
TRUE
WRITELN (' OP? ERATOR: TAG: ' , RATORT , ' OPCOUNT: ' , RATORN , ' OPERATOR: ' , OPERATOR , ' PSOT: ' , RATORP)
FALSE
WRITELN (' OP? ERAND: D: ' , D , ' T1: ' , T1 , ' Q: ' , Q , ' T2: ' , T2)

Figure 5

THE APPLICATION OF SOFTWARE ENGINEERING TECHNIQUES TO THE DESIGN OF RELATIVELY MACHINE-INDEPENDENT CODE GENERATORS*

Robert E. Noonan
College of William and Mary

Patricia Timpanaro
Computer Sciences Corporation

A serious problem in providing high level language support for embedded computer systems is the ability to construct code generators for these machines quickly, cheaply, and reliably. At the current time, HAL/S is the NASA standard programming language for flight software. The first phase of the HAL/S compiler produces an intermediate code called HALMAT, which is basically in the form of triples. A cross-compiler for HAL/S can be constructed by combining the first phase of the compiler with a HALMAT-to-machine-language translator. The problem is to build these code generators so that as much of the code generator as possible is reusable in moving from one machine to another.

The research undertaken was to apply the techniques of software engineering to the design of such code generators. The software design methodologies used included: the Jackson design methodology, hierarchical machine design, information hiding, composite design, and iterative enhancement.

The initial design combined the notions of information hiding and hierarchial machine design in such a way that each level of the design tries to hide some design decision from the levels above it. Specifically, this approach was applied to hide the details of the architecture of the machine; that is, two one-address, single accumulator machines would have large portions of the code generator identical. Jackson's methodology was used for the design of the input (HALMAT) routines. A complete implementation of a minimal HAL subset was implemented. As additional features were implemented, the overall design and modularity of the code generator was reviewed and improved.

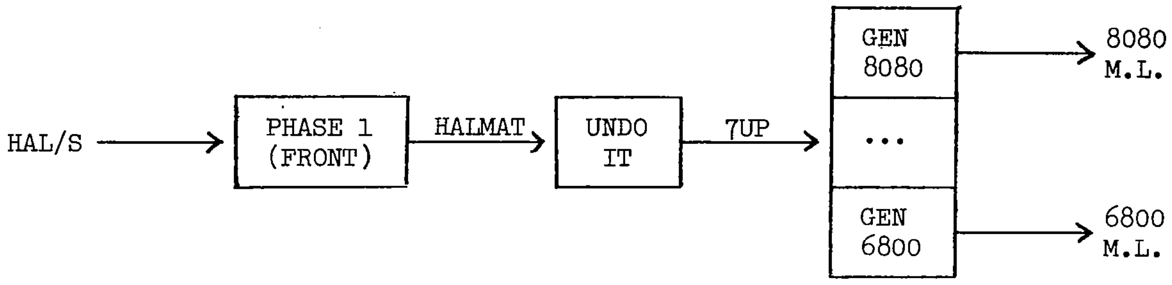
The current design (essentially, the second iteration) consists of a two phase mapping. First, HALMAT is translated to a hypothetical, machine language known as 7UP; this machine has a single accumulator, no index register, and one-address instructions. The second phase maps 7UP to machine language, in our case, an Intel 8080.

Most of the effort resides in the HALMAT to 7UP translation. This process need only be done once for all machines of similar architecture. In this case, only the 7UP to machine code translator needs to be rewritten.

At the current time, research is concentrating on the development of a software tool to simplify the complex case analysis needed to generate good code. This has resulted in the development of a non-procedural, problem-oriented programming language called CGGL (Code Generator Generator Language). The output from a CGGL compilation is a PASCAL program for generating machine code from an intermediate code. Preliminary results from the use of this tool are very encouraging.

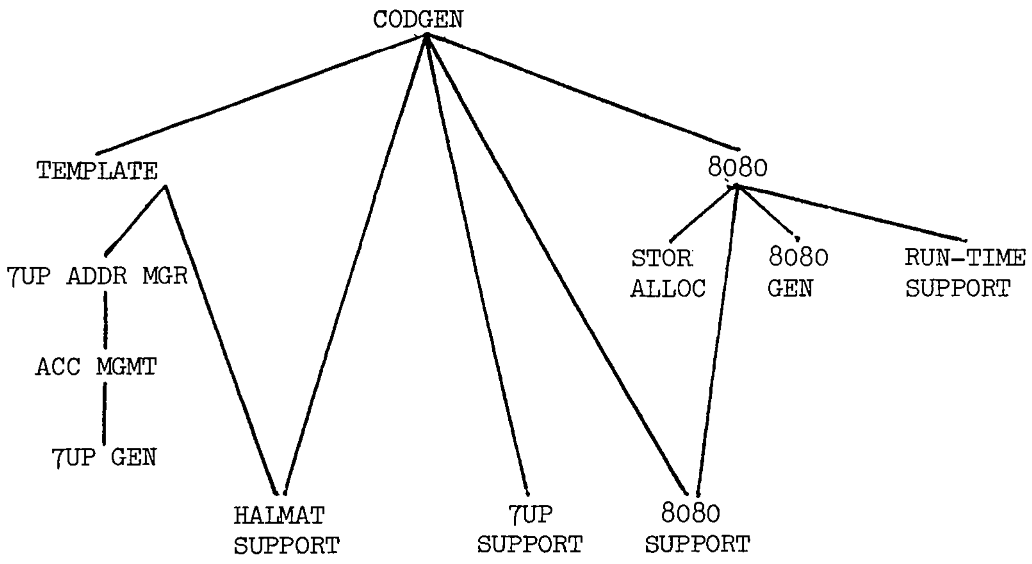
---

* This work performed under NASA contracts NAS 1-14972 and NAS 1-14900.



HAL Compilation System

Figure 1



Logical Structure of the Code Generator

Figure 2

Class No.	Class No. / Class Name	1	2	3	4	5	6	7	8	9	10
1	8080 Support	12							1		
2	7UP Support										
3	HALMAT Support			2					3		
4	Accumulator Management			2	1		2				
5	Storage Allocator	1							1		
6	7UP Generator						1		1		
7	8080 Generator	5	1					1	1		
8	Code Generator	3	1	1		3		2	1	1	
9	Miscellaneous			3							
10	Template			6	3		1		1		1
11	7UP Address Manager			2	2		1				

Fanout of Module Classes

Figure 3

Notation: ACC = accumulator PC = program counter
EA = effective address C(...) = contents of ...

Opcode	Mneumonic	Interpretation
0	STORE	C(ACC) → C(EA)
1	LOAD	C(EA) → C(ACC)
3	JUMP	EA → C(PC)
4	JFALSE	<u>if</u> C(ACC) = <u>false</u> <u>then</u> EA → C(PC)
6	CALL	subroutine call
7	RETURN	return from a subroutine
8	ADD	C(ACC) + C(EA) → C(ACC)
11	CNEQ	(<u>if</u> C(ACC) ≠ C(EA) <u>then</u> <u>true</u> <u>else</u> <u>false</u>) → C(ACC)
18	SUBSCR	C(ACC) + EA → C(ACC)
19	SAVE_ADDR	C(ACC) are saved in an address temporary
20	LABEL	associate the PC with the address field

Sample 7UP Operations

Figure 4

HAL/S A = A + 1 ;

HALMAT

Operation	Operand 1	Operand 2
IADD	SYT,5	LIT,3
IASGN	VAC,*-1	SYT,5

7UP

Operation	Tag	Entry
LOAD	SYT	5
ADD	LIT	3
STORE	SYT	5

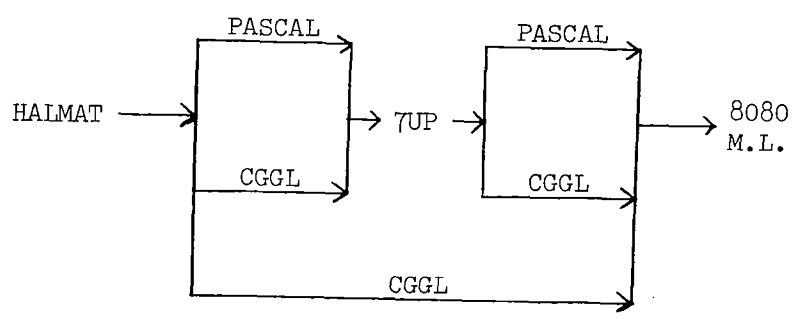
Given:

	SYT Table
5.	A
6.	...

	LIT Table
3.	1
4.	...

Sample Translation to 7UP

Figure 5



Possible Uses of CGGL

Figure 6

AN AVIONICS SOFTWARE DEVELOPMENT EXPERIENCE

L. C. Klos
General Dynamics Corporation

The Fire Control Computer on the USAF F-16 aircraft provided the opportunity and requirement for a new avionics software development program. The Fire Control Computer provides weapon computations, pilot interface, and system integration functions for the avionics system. The computer is required to manage an external data bus and support real-time multiprogrammed applications at various execution rates. It is programmed in the J3B-2 dialect of the JOVIAL language under stringent memory and execution time usage restrictions.

The presentation covers the impact which these constraints had upon the developing software and upon the tools which were used to support that development. The tools used include a graphical aid to software partitioning, an interface management data base, an interface data base processor which automatically generates JOVIAL common data areas, a source code indenter, a standards checker, an IBM 370 host computer debug facility, a JOVIAL flowcharter, configuration management procedures with automated support, and a hot-bench dynamic test station for flight simulation and functional checkout. These individual tools met with varying degrees of success and required varying degrees of effort in implementation. Directed flowgraphs and associated computerized interface management for example, were relatively difficult to use but were judged to be worthwhile and successful. Use of these techniques materially aided partitioning and structuring of the software in the design stages and helped to reduce implementation and checkout time. The hot-bench dynamic test station and other laboratory facilities were also quite successful. Real-time simulation of the aircrafts external environment and pilot interaction with the simulated system allowed realistic tests of the integrated software in a controlled and observable environment. The standard module option/host debug facility, however, required significant effort in use but was much less successful. This facility allowed modules to be written in a standard form with all input and output data provided as formal parameters in the module calling sequence to facilitate checkout on the IBM 370 host computer. A later step converted selected formal arguments into common data area parameters for implementation efficiency.

The presentation also covers the apparent future of avionics software from a tool requirement viewpoint. The proliferation of microprocessors into avionics causes problems in communication and execution management, as well as in maintainability. The tools used on the current F-16 program in most respects will be suitable for future systems. Partitioning of software and interface management will become increasingly important as avionics systems become more integrated and computerized. Both totally software and hot-bench simulations will find increasing use in the modeling and testing of candidate system architectures.

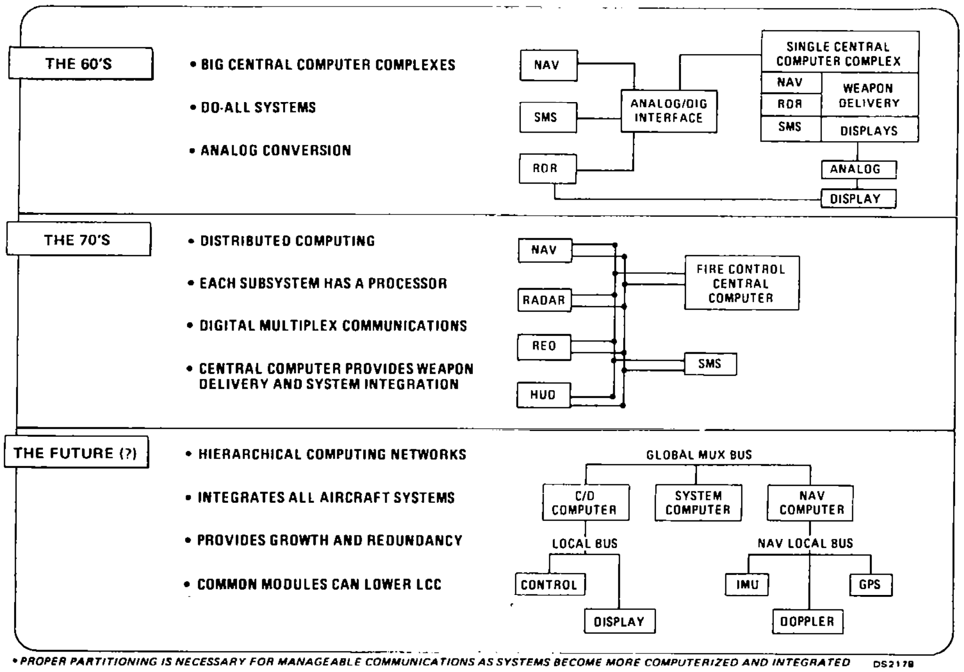
SYSTEMS INTEGRATION LABORATORY (SIL)

➧ THE SIL CONTAINS ACTUAL AVIONICS SYSTEMS HARDWARE IN A MINICOMPUTER SUPPORTED ENVIRONMENT. SPECIFIC SYSTEMS AND CAPABILITIES PROVIDED ARE:

- DYNAMICS TEST STATION (DTS)
  - HARRIS COMPUTER BASED SIMULATION OF AIRCRAFT AND ENVIRONMENT
  - ACTUAL AVIONICS HARDWARE IS EXERCISED
  - REAL-TIME INTERFACE DATA RECORDING IS PROVIDED
- AVIONICS EQUIPMENT BAY (AEB)
  - SIMPLIFIED AIRCRAFT SIMULATION USING AIRCRAFT COCKPIT SECTION
  - EVALUATES AVIONICS IN ACTUAL SPACE, ELECTRICAL AND COOLING ENVIRONMENT
  - EVALUATES SOFTWARE WITH REAL RADAR AND OPERATOR CONSTRAINTS
- REAL-TIME MONITOR UNIT (RTMU)
  - PROVIDES DATA RECORDING OF INTERNAL COMPUTER PARAMETERS
  - UNIQUELY IDENTIFIES DATA AND TIME OF RECORDING

Figure 1

TRENDS IN AVIONICS SOFTWARE ARCHITECTURE



THE 60'S
• BIG CENTRAL COMPUTER COMPLEXES
• DO-ALL SYSTEMS
• ANALOG CONVERSION
NAV
SMS
RDR
ANALOG/DIG INTERFACE
SINGLE CENTRAL COMPUTER COMPLEX
NAV
RDR
SMS
WEAPON DELIVERY
DISPLAYS
ANALOG
DISPLAY
THE 70'S
• DISTRIBUTED COMPUTING
• EACH SUBSYSTEM HAS A PROCESSOR
• DIGITAL MULTIPLEX COMMUNICATIONS
• CENTRAL COMPUTER PROVIDES WEAPON DELIVERY AND SYSTEM INTEGRATION
NAV
RADAR
REO
HUD
FIRE CONTROL CENTRAL COMPUTER
SMS
THE FUTURE (?)
• HIERARCHICAL COMPUTING NETWORKS
• INTEGRATES ALL AIRCRAFT SYSTEMS
• PROVIDES GROWTH AND REDUNDANCY
• COMMON MODULES CAN LOWER LCC
GLOBAL MUX BUS
C/D COMPUTER
SYSTEM COMPUTER
NAV COMPUTER
LOCAL BUS
NAV LOCAL BUS
CONTROL
DISPLAY
IMU
GPS
DOPPLER

• PROPER PARTITIONING IS NECESSARY FOR MANAGEABLE COMMUNICATIONS AS SYSTEMS BECOME MORE COMPUTERIZED AND INTEGRATED DS2178

Figure 2

F-16 ARCHITECTURE OVERVIEW

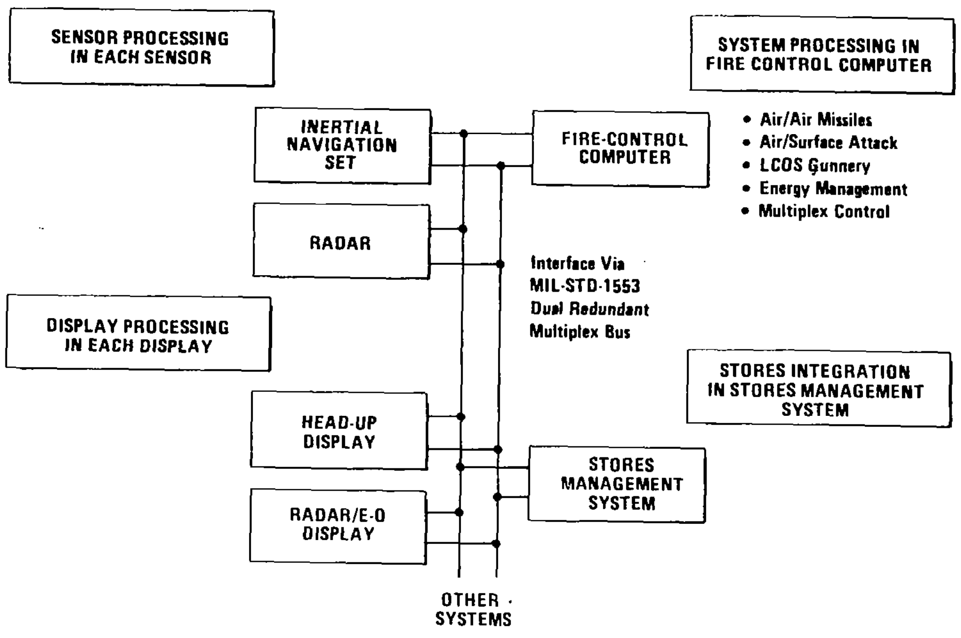

SENSOR PROCESSING IN EACH SENSOR
SYSTEM PROCESSING IN FIRE CONTROL COMPUTER
INERTIAL NAVIGATION SET
FIRE-CONTROL COMPUTER
• Air/Air Missiles
• Air/Surface Attack
• LCOS Gunnery
• Energy Management
• Multiplex Control
RADAR
Interface Via MIL-STD-1553 Dual Redundant Multiplex Bus
DISPLAY PROCESSING IN EACH DISPLAY
STORES INTEGRATION IN STORES MANAGEMENT SYSTEM
HEAD-UP DISPLAY
STORES MANAGEMENT SYSTEM
RADAR/E-O DISPLAY
OTHER SYSTEMS

Figure 3

F-16 SOFTWARE DEVELOPMENT CONTEXT

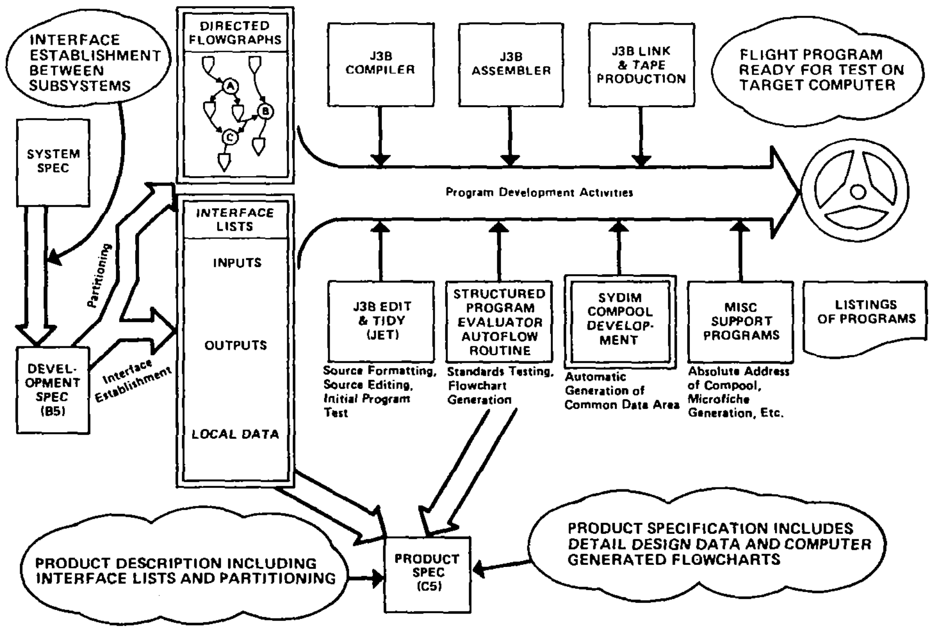

INTERFACE ESTABLISHMENT BETWEEN SUBSYSTEMS
DIRECTED FLOWGRAPHS
J3B COMPILER
J3B ASSEMBLER
J3B LINK & TAPE PRODUCTION
FLIGHT PROGRAM READY FOR TEST ON TARGET COMPUTER
SYSTEM SPEC
Program Development Activities
INTERFACE LISTS
INPUTS
OUTPUTS
LOCAL DATA
Partitioning
Interface Establishment
DEVEL-OPMENT SPEC (B5)
J3B EDIT & TIDY (JET)
Source Formatting, Source Editing, Initial Program Test
STRUCTURED PROGRAM EVALUATOR AUTOFLOW ROUTINE
Standards Testing, Flowchart Generation
SYDIM COMPOOL DEVELOP-MENT
Automatic Generation of Common Data Area
MISC SUPPORT PROGRAMS
Absolute Address of Compool, Microfiche Generation, Etc.
LISTINGS OF PROGRAMS
PRODUCT DESCRIPTION INCLUDING INTERFACE LISTS AND PARTITIONING
PRODUCT SPEC (C5)
PRODUCT SPECIFICATION INCLUDES DETAIL DESIGN DATA AND COMPUTER GENERATED FLOWCHARTS

Figure 4

TOOLS USED IN THE DEVELOPMENT

TOOL	DESCRIPTION	DEGREE OF SUCCESS
• ATMS	WORD PROCESSING SYSTEM	HIGH
• DIRECTED FLOW GRAPHS	GRAPHICAL DESIGN TECHNIQUE	HIGH*
• SYDIM	COMPUTERIZED INTERFACE MANAGEMENT	HIGH
• JET	JOVIAL EDIT AND TIDY – Source Formatting – Source Editing – Standard Module Option/Host Debug Facility	 HIGH MED LOW*
• SPEAR	STRUCTURED PROGRAMMING EVALUATION AND AUTOFLOW ROUTINE – Structured Programming Standards Checker – Flowchart Generation	 LOW HIGH
• AWIP	AUTOMATED WEAPON INITIALIZATION PROGRAM	HIGH
	ABSOLUTE ADDRESS GENERATION FOR COMPOOL	HIGH
	CONFIGURATION MANAGEMENT SYSTEM	HIGH
• JOVIAL J3B-2	HIGH ORDER PROGRAMMING LANGUAGE	HIGH
• DTS	COMPUTER SUPPORTED DYNAMIC TEST STATION	HIGH
• AEB	COMPUTER SUPPORTED AVIONICS EQUIPMENT BAY	MED
• RTMU	REAL-TIME MONITOR UNIT	MED
	COMPUTER FUNCTIONAL SIMULATOR	LOW
	REAL-TIME DATA RECORDING	HIGH
• COMPALL	MULTIPLE COMPILATION AND ASSEMBLY	HIGH

THESE ITEMS ARE SIGNIFICANTLY DIFFICULT TO USE

Figure 5

DIRECTED FLOWGRAPH WITH INTERFACE

```
NAVIGATION SUPPORT COMPONENT – INTERFACE SPECIFICATION
OUTPUT DATA SIGNALS

SIGNAL LBL   TYP  HZ  BLOCK  SIGNAL NAME
  RHOY        F   50  NS05   RHO Y
  RHOZ        F   50  NS05   RHO Z
**BEG CETOP         ARRAY
 ARRAY CETOP                  3  3
  CETOPXX     F   50  NS06   DIRECTION COS CXX–EARTH TO PLAT
  CETOPXY     F   50  NS06   DIRECTION COS CXY–EARTH TO PLAT
  CETOPXZ     F   50  NS06   DIRECTION COS CXZ–EARTH TO PLAT
  CETOPYX     F   50  NS06   DIRECTION COS CYX–EARTH TO PLAT
  CETOPYY     F   50  NS06   DIRECTION COS CYY–EARTH TO PLAT
  CETOPYZ     F   50  NS06   DIRECTION COS CYZ–EARTH TO PLAT
  CETOPZX     F   50  NS06   DIRECTION COS CZX–EARTH TO PLAT
  CETOPZY     F   50  NS06   DIRECTION COS CZY–EARTH TO PLAT
  CETOPZZ     F   50  NS06   DIRECTION COS CZZ–EARTH TO PLAT
**END CETOP
  WNDLNGTK    F    6  NS07   WIND LONGTRACK
  WNDCRSTK    F    6  NS07   WIND CROSSTRACK
  FN          F    6  NS07   LOCAL GRAVITY
  GNDTRK      AO  50  NS09   GROUND TRACK
  GNDSPD      F    2  NS10   GROUND SPEED
  TASXF       F   25  NS13   TRUE AIRSPEED FILTERED X
  TASYF       F   25  NS13   TRUE AIRSPEED FILTERED Y
  TASZF       F   25  NS13   TRUE AIRSPEED FILTERED Z
  DVALUE      F    6  NS14   'D' VALUE
  FALTPMAZ    AO  50  NS15   FLIGHT PATH MARKER AZIMUTH
  FALTPMEL    AO  50  NS16   FLIGHT PATH MARKER ELEVATION
```

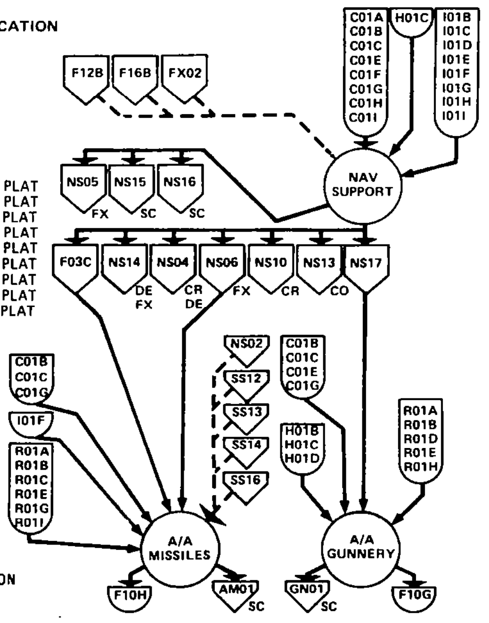


Figure 6

FUNCTIONAL SIMULATION OF SPACE SHUTTLE FLIGHT PROGRAMS

Arra Avakian
Intermetrics, Inc.

HAL/S is the computer programming language chosen by NASA for the Space Shuttle project. It was designed for space applications, and includes such features as vector-matrix arithmetic and real-time process control statements. Within the Shuttle programming context, HAL/S programs can execute in at least two distinct simulation modes as well as actual execution on a flight computer. The simulation mode closest to the flight computer involves compilation using the flight computer compiler (HAL/S-FC) followed by simulation on an interpretive computer simulator (ICS). Although a very close reproduction is attained, the usefulness of the ICS mode of simulation is limited by its high CPU costs to small scale simulations and compiler checkout. Another mode of simulation is available which operates at the level of a HAL/S statement. Terms used to describe this mode are "functional simulation" (FSIM) or "statement level simulation" (SLS). Whereas the smallest unit simulated without environmental interaction in the ICS mode is one machine instruction, the smallest unit in FSIM mode is one HAL/S statement. FSIM is not a bit-for-bit simulation, so it operates with much greater efficiency. However, its accuracy is good enough so that almost all of the flight software algorithm checkout is performed under this mode of simulation.

FSIM involves the use of another compiler with a common language analysis phase but a separate code generator, the HAL/S-360 compiler. Although the HAL/S-360 compiler generates 360 machine code, allowing execution to proceed at the full rate of the host 360/370 machine, interaction with a simulation monitor can occur at any HAL/S statement in a manner analogous to an ICS. Pseudo-real time can be maintained, so that interactions may occur at any desired time as well. A model of the Flight Computer Operating System (FCOS) is supplied by the compiler system, allowing full use of the real-time process control features of the language. The total effect is to simulate execution of the same HAL/S program compiled for and executing on the real flight computer under control of its operating system.

The key to the simulation monitor's control of HAL/S execution at the statement level is the "hook" instruction inserted between each statement by the HAL/S compiler. The "hook" instruction causes control to pass to a statement processor routine. This routine gives control to the monitor whenever conditions occur which have been previously established. Such conditions may be the execution of a specified ("hot") statement, or the arrival of a specified pseudo-time. The statement processor advances the pseudo-time by a time cost computed by the compiler for each statement.

HAL/S-360 COMPILER SYSTEM OPERATION

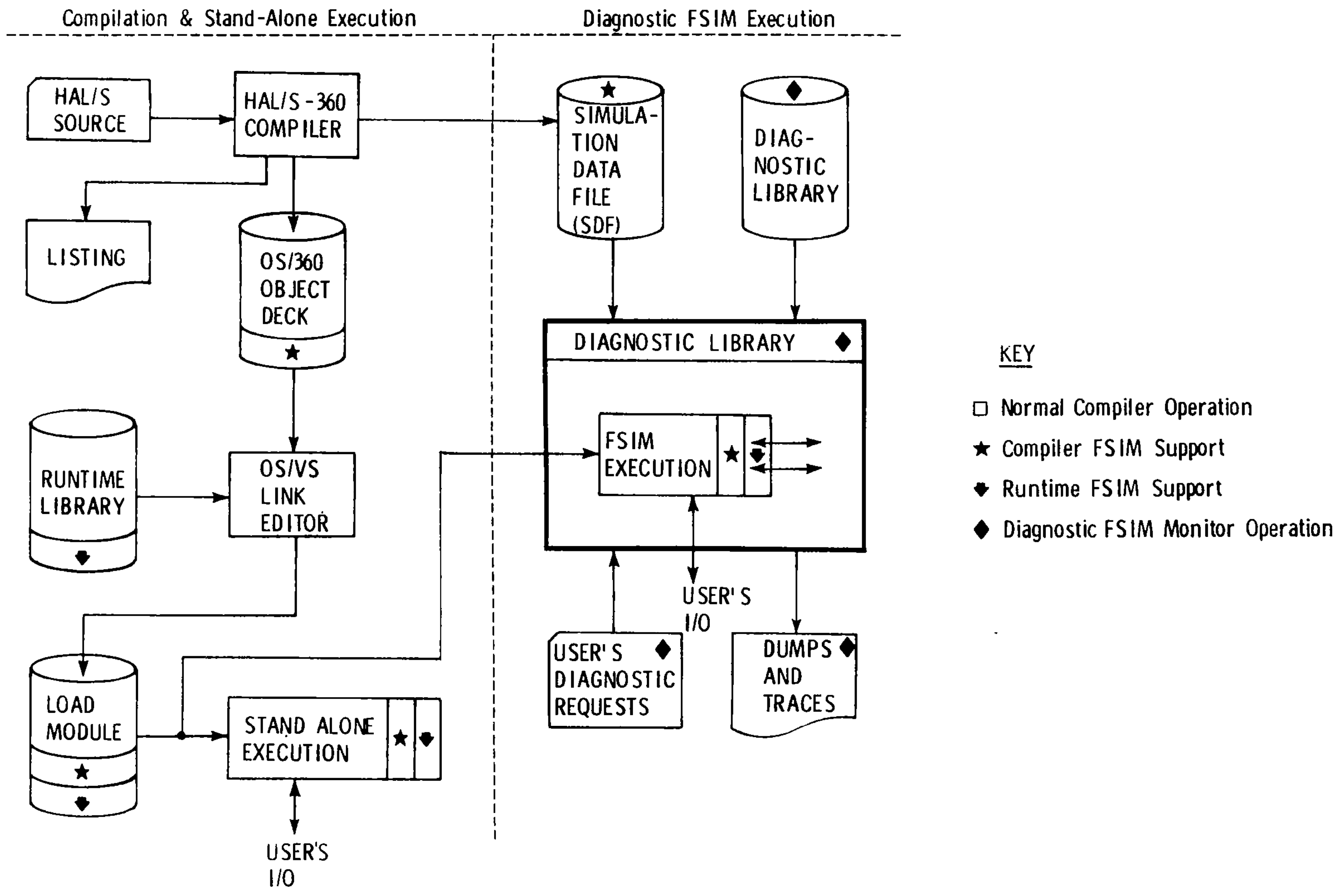


Figure 1

FSIM SUPPORT FUNCTIONS

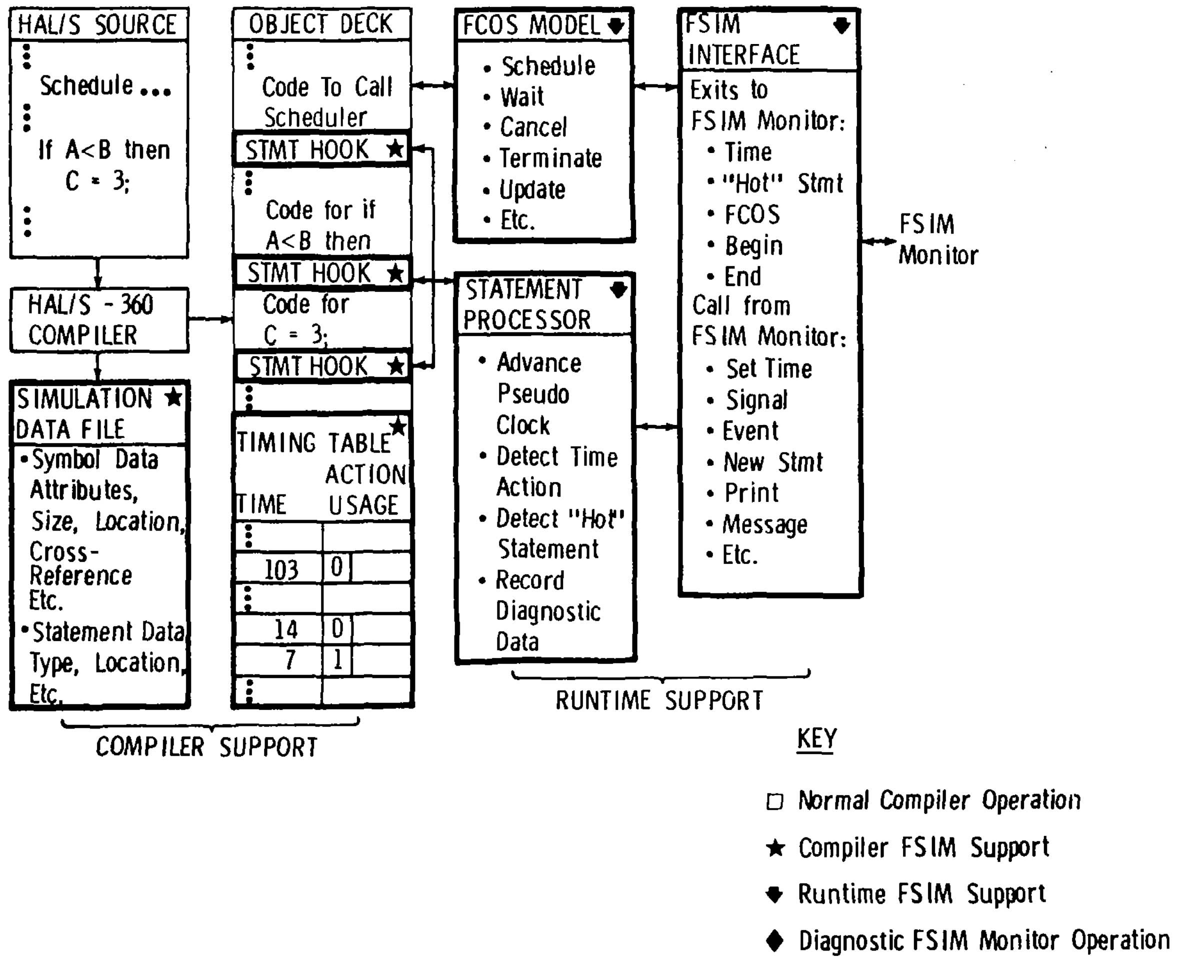


Figure 2

DYNAMIC STATEMENT PROCESSOR

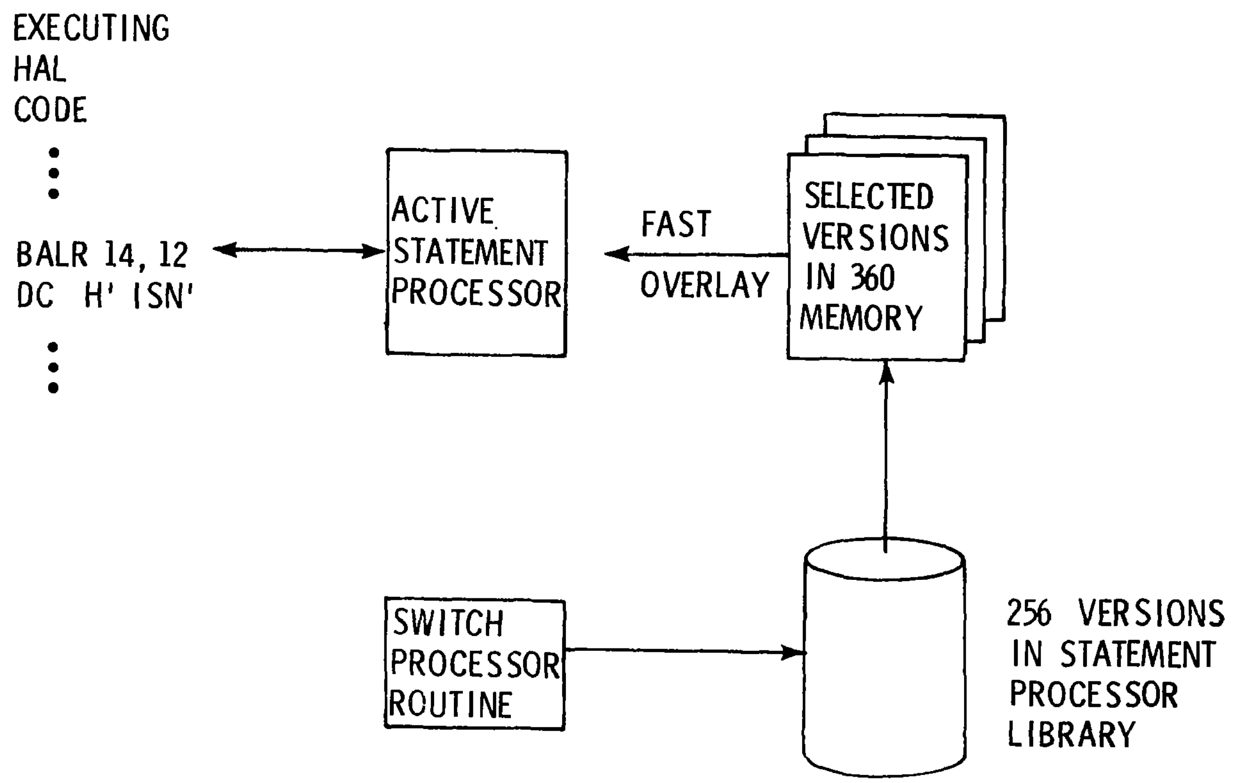

EXECUTING
HAL
CODE
BALR 14, 12
DC H' ISN'
ACTIVE STATEMENT PROCESSOR
FAST
OVERLAY
SELECTED VERSIONS IN 360 MEMORY
SWITCH PROCESSOR ROUTINE
256 VERSIONS IN STATEMENT PROCESSOR LIBRARY

Figure 3

PROVE

A Tool For Software Verification

Randall J. Varga
The Singer Company
Kearfott Division

ABSTRACT

PROgram Verification Equipment (PROVE) is a dynamic simulator used at Singer-Kearfott in the development of Advanced Avionic Software Systems. It is currently being used in the verification of the calibration and alignment algorithms for the Space Shuttle Inertial Measurement Unit (IMU).

PROVE was developed to provide a low cost controlled environment for the verification of calibration algorithms and associated systems. The PROVE simulator is a significant improvement over previous techniques which were hampered by project unique hardware which interfaced the simulator with the operational system.

INTRODUCTION

As today's processors become more powerful, advanced airborne avionic systems are required to perform more tasks at faster rates then ever before. These additional functions mandate that software systems become increasingly large and complex, requiring more extensive checkout in order to verify satisfactory performance of their tasks during all operational phases. In fact, the complexity of the checkout required to verify the software increases exponentially with the size of the program (see figure 1). Thus finding an automatic timely method of verifying software is also an exponentially growing problem.

At Singer-Kearfott software is verified by a multiple step process. First, the entire software package is designed in a Top Down Structured manner. This allows for a cohesive interrelationship of modules. This is then followed by extensive desk debugging of the individule modules. This includes static simulation on a large scale general purpose computer system. Considering the typical iteration rate for avionic systems (5Ø-2ØØHz), and the length of execution (upwards to 12 hours) the cost of fully simulating the software on the large scale computer becomes increasingly high. In light of the cost factor, software modules should not be fully checked out by this method. The next step in verification prior to flight testing of the software package is dynamic simulation of the entire system. To facilitate this dynamic checkout a series of simulation systems have been developed by Singer-Kearfott over the years. These include a Digital Inertial Measurement Unit Dynamic Simulator (DIMUDS) (ref 1), Communication And Navigation Dynamic Simulator (COMMANDS) (ref 2), as well as PROgram Verification Equipment (PROVE).

OBJECTIVES OF PROVE

In designing and building PROVE, several objectives were established. These objectives were:

(1) The simulation must run in a real time environment.
This requirement detects timing unique errors in the software modules. The types of errors which are isolated by this requirement are subroutine re-entrancy, cycle interference, and system time overload.

(2) The entire software package must be tested.
The necessity for this requirement is obvious. It mandates that every software module be fully verified prior to final system generation. Thus it prevents untested software going undetected merely because it is not needed for simulation checkout.

3) The interface format must be the same as with the actual hardware.
This requires the full verification of all sensor unique routines. All reformatting and scaling of the sensor data are performed and verified. It also alleviates the necessity of generating simulator unique interface routines merely because the interface format is different than the real hardware.

(4) The simulator must be cost effective to operate.
This is a double requirement. First the simulator must be inexpensive to build, operate, and modify. In addition, it should use commercial equipment as much as possible, eliminating the need for specialized test equipment and hardware. Second, simulation must not require an excessive amount of time to perform. That is, if due to model constraints, the simulator cannot be made to run in real time every effort must be made to minimize the excess of real time.

(5) The simulator must accurately model the actual hardware. This requires that given the same output from the operational system the simulator and the hardware must produce comparable results. There must be nothing known that the simulator does not model. Conversly there must be nothing modeled which does not occur in the actual hardware.

To implement the PROVE concept Singer-Kearfott has chosen to use the Hewlett Packard 2100 mini computer. A significant factor in making this selection is the fact that the operational computer to which the PROVE was initially connected was also a HP2100 computer. This significantly simplifies the interfacing problem. The current implementation is shown in figure 2.

ADVANTAGES TO PROVE

Simulation approaches similar in nature to PROVE have been used at Singer-Kearfott for many years with great success. PROVE itself is being used on several projects currently under development, the most notable of which is the verification of the Space Shuttle IMU calibration algorithms. These projects have enjoyed many advantages over those that did not use PROVE. Some of these are described below:

Verification of an operational software system employing PROVE, accuratly simulates the environment in which the system must operate. The digital interface between the operational system and the sensor hardware is modeled to be consistant with the actual hardware interface. This includes the data which are transmitted as well as their format. This reproduction of the interface allows the operational system to be fully verified. This not only includes the data reduction portion of the system but also the reformatting of the interface data which are required when the actual hardware is employed. In addition the software unique to the simulator is for the most part non-existant. The only module required is that which actually performs I/O between the simulator and the operational software system. With the exception of the true I/O routine which is checked out by means of static simulation and actual hardware runs, the entire operational software system has been verified before ever having been run with the actual sensor hardware.

A further advantage of employing PROVE is that the entire software system is dynamically verified in a real time environment. This is possible because the PROVE creates the real time environment in which the actual system will operate. This is accomplished by employing a real-time clock in the PROVE computer as a time referance. After the prescribed amount of time has elapsed the PROVE software causes an interrupt to occur in the operational computer by means of a dedicated I/O instruction. This interrupt is seen by the operational system as if originating in the actual sensor hardware and is treated in exactly the same manner. Thus the real-time features of the operational system are preserved and verified.

Another major advantage in using PROVE is that it is a cost effective means of software verification. There are several factors about the PROVE which make it cost effective. Among these is that all the hardware components which comprise PROVE are commercially available. This allows for the construction of PROVE in a timely fashion. There is no long lead time required for hardware development. The design costs of the system are greatly reduced since there are no unique hardware modules which must be designed and built.

As an illustrative example, let us examine the problems encountered by a typical project that did not employ the PROVE concept of software verification. This project's requirements were to control a sensor, process the data, and communicate with another computer. The interface with the other computer was over a parallel data line. For software verification purposes it was decided to use a specially designed hardware interface module (see figure 3) . This particular module had a lead time near that of the system development time. The obvious result was that the intercomputer communication software could not be verified until near the end of the project. In addition since the communication requirement was crucial to the system, there was strong schedule pressure to shorten the development time of the hardware. As a result of these problems and pressures the module was delivered late and required the design engineer to maintain the equipment. Further the module proved to be unreliable and frequently failed. The net result was that the project significantly increased both cost and schedule projections. Similar projects using the PROVE were delivered on time and at projected cost.

Since PROVE is built from commercially available equipment, there is no need to train personnel in it's maintainence since this can be readily obtained from the component manufacturers by means of service contracts. Another major factor in making PROVE cost effective is that if the components used in assembling the system are purchased, then there is only the initial outlay of funds to purchase the equipment and no periodic rental charges. The equipment would be continuously available for project use without recurring cost.

Since the PROVE system has been assembled using commercially available components it was decided not to employ simulator unique interface hardware. The communication between the simulator and the operational system is carried out over the standard I/O interface cards available from the manufacturer. This allows the I/O interface to be completely modified by simply changing the software interface routines. This makes the PROVE useable to other projects without having to reconfigure hardware modules. It also reduces the length of time required to change from one project to another. Project change is no more complex than reloading the computer and connecting another cable to the interface card (approximatly 5 minutes).

Since the initial development of PROVE several enhancements and improvements have been made to the system. The most notable of these enhancements is the inclusion of failure simulation. It should be noted that in order to verify failure detection software using the end item a mechanism of artifically inducing failures at the system level is required. This is generally a difficult task and often requires hardware modification. This is accomplished in the PROVE computer by modifying the output as computed by the sensor model in a prescribed manner. This was mechanized by adding routines between the data formatting routines and the I/O routine (see figure 4). The setting of these failures is under operator control from the system input device. The operator selects the error he wishes to simulate and the time duration of the failure. The system then perturbs the output for the selected length of time. Since the modeling of the failure occurs after the formatting of the output data, there is no feedback to the sensor model and therefore no modification of the system output after the failure simulation is finished. In this manner intermittant hardware errors and hardware noise are simulated. This therefore verifies the error recovery modules in the system.

PROBLEMS WITH SPACE SHUTTLE PROVE

PROVE has been used at Singer-Kearfott for several years with good results. Thus far there have been no conceptual problems encountered. The only problem areas which have been encountered were peculiar to the particular implementation. These were mainly limitations imposed by the architecture of the mini-computer used. As a result of these problems some trade off studies were performed on the design criteria for the particular implementaion. Because of these studies, certain design goals were relaxed in order to produce a useable working system in a timely fashion.

One of the first problems faced by the designers was the need for high precision in the computations. Because of the iteration rate and the length of execution of the programs it was determined that the mini-computer floating point format was not adequate. Errors in the computation due solely to round off were deemed unacceptable. For this reason it was determined that certain critical sections of the sensor model had to be written in extended precision arithmetic. This had the disadvantage of both increasing size and lengthing the execution time of the model. The increase in memory was minor (less than 100 words of memory). However the increase in execution time had a more significant impact on the system. Because of the increased execution time the simulator was unable to be made to run in real time. In fact the simulator has been forced to run 25% slower than real time.

Since the simulator has been forced to run slower than real time, certain error conditions may remain untested. The primary condition which may go undetected is system time overload. This is because the operational system is allowed 25% additional time to complete it's assigned tasks. Therefore to detect a time overload condition in the normal sense the system would have to be greater then 25% overloaded. This problem has been alleviated by not permitting the time loading of the operational system to increase above 80%.

Another problem resulting from the slower execution of the simulator is the fact that the operational system will not be interrupted at the same point during simulation as when running with the actual hardware. This may allow non-interruptable routines to go undetected. The possibility of this type of error remaining in the delivered system was eliminated because the final step in software verification always includes full verification testing with the actual hardware, and at that point the error would be detected.

FUTURE OF PROVE

PROVE has proven itself to be an invaluable tool for software verification. It has allowed for the controlled and detailed testing of complex software systems without having to resort to costly flight testing. There are however certain areas of improvement which can be made to the system. The most obvious improvement is the decreasing of the execution time of the sensor model. This it appears can be accomplished in a fairly straight forward manner by microcoding selected fundamental functions of the model. The types of functions which would be microcoded are matrix multiply and sine/cosine functions. This would not impact the ability to make future improvements to the system since only the most fundamental functions would be selected for microcode.

REFERENCES

1. Frisina, J. N.: Dynamic Simulation via Minicomputers. Proceedings of 7th Annual Pittsburg Conference on Modularity and Simulation, Apr. 1977, p. 477.

2. Frisina, J. N.; Steele, W. J.; and Schlenger, J. I.: Dynamic Simulation of a Multi-Sensor Communication and Navigation System. Proceedings of NATO AGARD Conference on Navigation and Gùidance, May 1978.

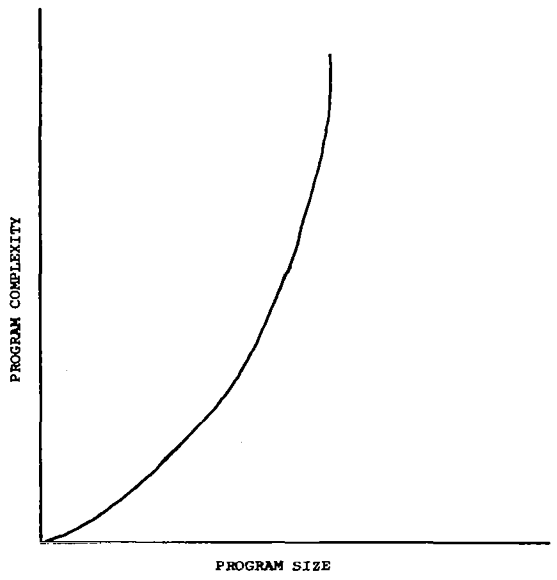


Figure 1

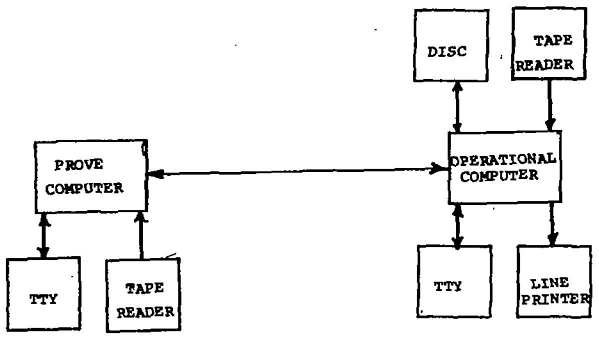


Figure 2

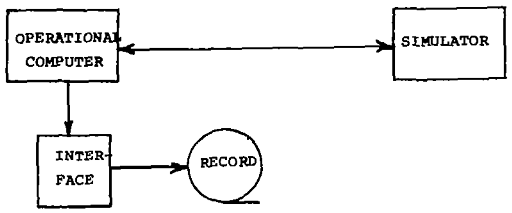


Figure 3

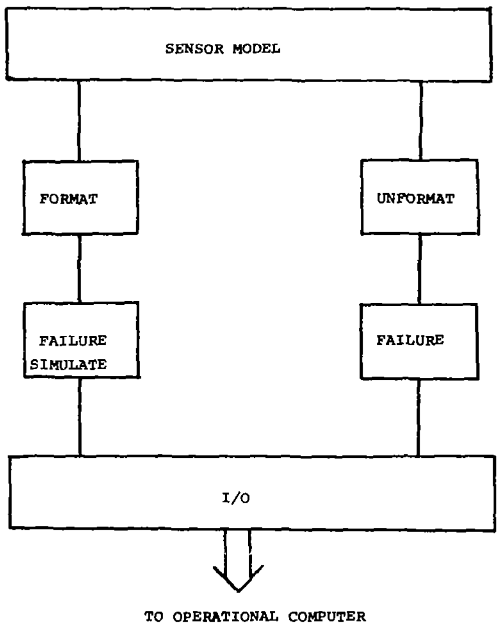


Figure 4

INTERPLANETARY SPACECRAFT COMPUTER SOFTWARE TEST AND VALIDATION TOOLS*

Daniel E. Erickson
Jet Propulsion Laboratory

Several characteristics of unmanned interplanetary space missions impose unique requirements upon the embedded computer systems which control the spacecraft and the tools used to validate the programming of these computers. The long mission duration creates a need for reprogramming after launch to accommodate evolving scientific requirements, a better understanding of the environment, or spacecraft anomalies. The stringent mass/volume/power limitations, coupled with a desire to maximize the information return mandates efficient, assembly language coding and detailed simulations of computer execution through the entire mission. The simulation must be much faster than real time. This simulation has traditionally been performed in software on large mainframe. Throughput times and costs which are proportional to the efficiency of the simulation, the number of processors being simulated, and the activity of each processor have grown due to increased processor activity. Current Voyager simulation operation is costing $4500 per week.

A hardware accelerated simulation tool (HAST) has been developed for Voyager. It is in the final stages of acceptance testing and will be phased into operations following Jupiter Encounter. The major problems which were solved in the design were the provision of visibility into the execution of the flight software via memory traces and the achievement of the desired 100 to 1 speedup ratio.

For the Galileo project, no detailed simulator in software is planned. Simulation of the seven RCA 1802 microprocessors, running in the active state over 10 percent of the time would be prohibitively expensive. A hardware accelerated simulation tool is required for software development as well as software test and command sequence verification. This will require increased visibility into software execution, dictating a slowdown or suspend mode to give time for the printing of trace data. Furthermore, if significant speedup is required (greater than 20 to 1), functional simulation of the embedded software by hardware will be dictated. Some speedup and added visibility will be achieved by the emulation of the RCA 1802 with AM 2900 bit sliced logic.

* This work is being conducted at the Jet Propulsion Laboratory of the California Institute of Technology by agreement with the National Aeronautics and Space Administration under Contract NAS 7-100.

VOYAGER SEQUENCING PROCESS

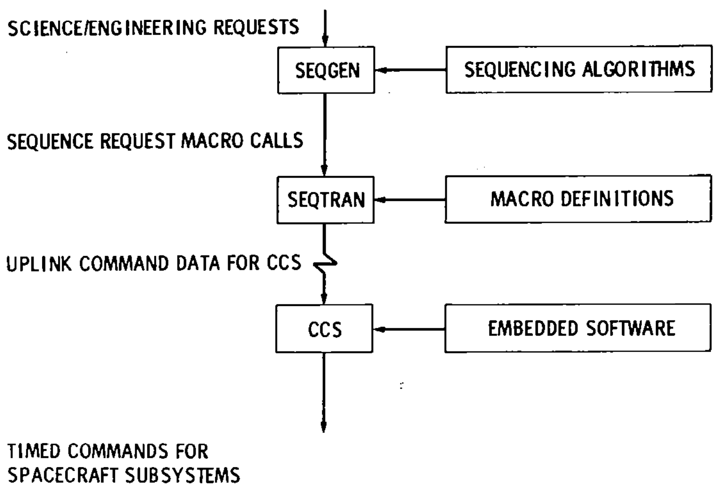

SCIENCE/ENGINEERING REQUESTS
SEQGEN
SEQUENCING ALGORITHMS
SEQUENCE REQUEST MACRO CALLS
SEQTRAN
MACRO DEFINITIONS
UPLINK COMMAND DATA FOR CCS
CCS
EMBEDDED SOFTWARE
TIMED COMMANDS FOR SPACECRAFT SUBSYSTEMS

Figure 1

VOYAGER SPACECRAFT DATA SYSTEM

(SIMPLIFIED)

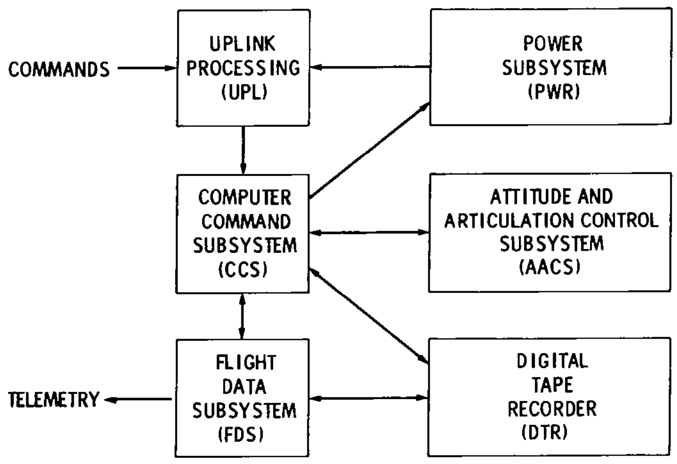

COMMANDS
UPLINK PROCESSING (UPL)
POWER SUBSYSTEM (PWR)
COMPUTER COMMAND SUBSYSTEM (CCS)
ATTITUDE AND ARTICULATION CONTROL SUBSYSTEM (AACS)
TELEMETRY
FLIGHT DATA SUBSYSTEM (FDS)
DIGITAL TAPE RECORDER (DTR)

Figure 2

OTHER HAST CONSTRAINT CHECKS

- INTERRUPTS DISABLED FOR LONGER THAN 100 msec
- INTERRUPT MISSED DUE TO SLOW PROCESSING
- OUTPUTS TOO CLOSE TOGETHER FOR RECEIVING SUBSYSTEM
- INVALID OUTPUT SEQUENCE

Figure 3

VOYAGER SIMULATION REQUIREMENTS

- SIMULATE CCS EXECUTION OF EACH COMMAND SEQUENCE
  - MUCH FASTER THAN REAL TIME
  - SIMPLE OPERATION
  - SAVE/RESTART CAPABILITY
  - RESPONSIVE SIMULATION OF INTERFACING SUBSYSTEMS
- CHECK CONSTRAINTS ON USE OF EMBEDDED SOFTWARE
  - MEMORY TRACE
  - TIME OUT TESTS
- ADAPT TO CHANGES IN CCS SOFTWARE DETECT ERRORS IN CCS SOFTWARE
  - DETAILED SIMULATION OF CCS

Figure 4

PROBLEM OF EMBEDDED SOFTWARE VALIDATION FOR UNMANNED INTER PLANETARY SPACECRAFT CONTROL

- SPACECRAFT ADAPTABILITY - REQUIRES REPROGRAMMING WITH QUICK VALIDATION TURN-AROUND
- OPTIMAL USE OF SPACECRAFT RESOURCES - REQUIRES TIGHT CODE - ASSEMBLY LANGUAGE OR A UNIQUE SPECIAL PURPOSE HIGHER ORDER LANGUAGE
- CONTROL TASK COMPLEXITY - ELIMINATES THE POSSIBILITY OF DEVELOPING SUFFICIENT CONFIDENCE IN THE FLIGHT SOFTWARE WITH A MANAGEABLE SET OF TEST CASES
- SOLUTION - DETAILED MUCH FASTER THAN REAL TIME SIMULATION OF ALL FLIGHT SEQUENCES - HARDWARE ACCELERATED SIMULATION

Figure 5

HAST TIME ACCELERATION EXPERIENCE

• FASTEST POSSIBLE ACCELERATION	288:1	
• SLOWEST HARDWARE ACCELERATION	2:1	
• FASTEST OBSERVED ACCELERATION	180:1	(0.2% ACTIVE)
• SLOWEST OBSERVED ACCELERATIONS		
• DURING CHECKSUM	2:1	(100% ACTIVE)
• DURING UPLINK	20:1	(5% ACTIVE)
• DURING BUSY SEQUENCES	60:1	(1.3% ACTIVE)
• AVERAGE ACCELERATION FOR TYPICAL SEQUENCES (SIMULATION PHASE)	100:1	(0.6% ACTIVE)

Figure 6

HAST TIMING CONTROL

- TIMING SIGNAL (2.4 kHz ON SPACECRAFT)
  - CONTROLS TIMING ON BOTH THE MINICOMPUTER AND THE CCS EMULATOR
  - MAY BE IN ONE OF FOUR STATES:
    - REAL TIME (2.4 kHz)
    - X2 (4.8 kHz)
    - SPEED UP (614.4 kHz)
    - INHIBITED
  - SPEED UP ONLY ENABLED WHEN
    - BOTH CPUS ARE IN WAIT STATE
    - BOTH OUTPUT UNITS ARE AVAILABLE
    - THE MINICOMPUTER HAS COMPLETED ITS CALCULATIONS

Figure 7

HAST MEMORY TRACE OPTIONS
(5 - EXTRA BITS FOR EACH 18-BIT CCS MEMORY WORD)

- FETCH - IF WORD IS BEING EXECUTED AS AN INSTRUCTION
- READ - IF WORD IS BEING READ AS DATA
- WRITE - IF WORD IS BEING WRITTEN INTO
- WFETCH - IF WORD IS BEING EXECUTED AND MAY CAUSE A WRITE INTO ANY MEMORY CELL
- INHIBIT - IF THIS WORD IS EXECUTED, DISREGARD ALL OTHER OPTIONS

Figure 8

HARDWARE ACCELERATED SIMULATION TOOL SIMULATION PHASE

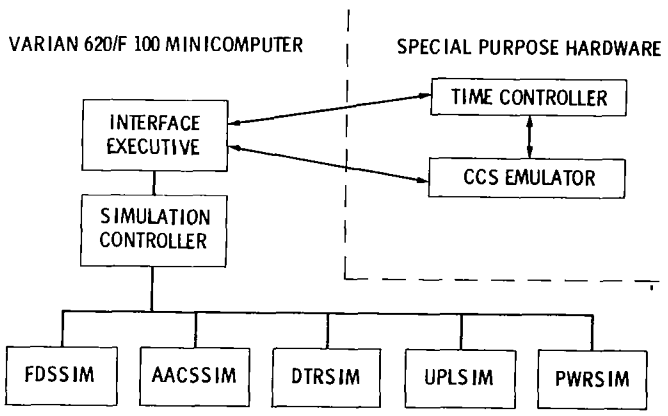


Figure 9

HAST SPEEDUP EXAMPLE

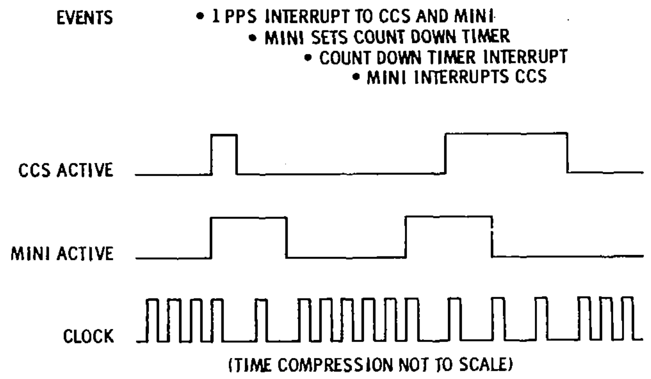

EVENTS
• 1 PPS INTERRUPT TO CCS AND MINI
• MINI SETS COUNT DOWN TIMER
• COUNT DOWN TIMER INTERRUPT
• MINI INTERRUPTS CCS
CCS ACTIVE
MINI ACTIVE
CLOCK
(TIME COMPRESSION NOT TO SCALE)

Figure 10

GALILEO HARDWARE ACCELERATED SIMULATION TOOL HARDWARE CONFIGURATION

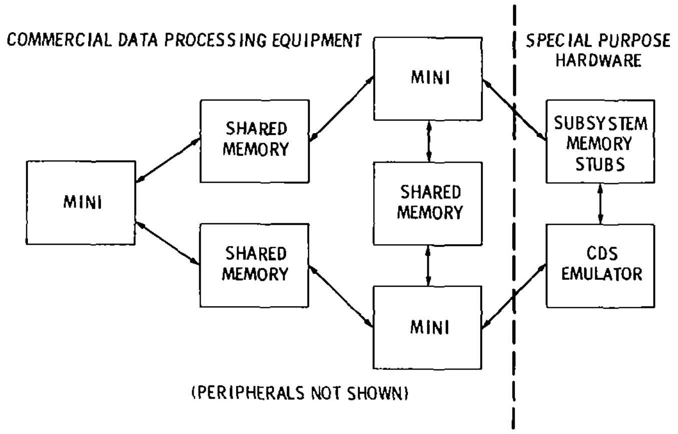


Figure 11

GALILEO SIMULATION REQUIREMENTS

- CDS EMBEDDED SOFTWARE DEVELOPMENT/TEST
  - DETAILED SIMULATIONS OF CDS MODULES
  - MEMORY TRACE FUNCTIONS
  - BREAKPOINTS
  - BUS TRAFFIC MONITORING
  - RESPONSIVE SIMULATION OF INTERFACING SUBSYSTEMS
- MISSION OPERATIONS SIMULATION OF COMMAND SEQUENCES
  - MUCH FASTER THAN REAL TIME
  - SAVE/RESTART CAPABILITY
  - SIMPLE OPERATION
  - INITIALIZATION FROM SPACECRAFT TELEMETRY

Figure 12

A SOFTWARE CHANGE DEVELOPMENT LABORATORY FOR SUPPORTING AN AGGREGATE OF EMBEDDED COMPUTER SYSTEMS

F. H. Kishi
TRW Defense and Space Systems Group

D. R. Corder
Oklahoma City Air Logistics Center

A concept, designated Software Change Development Laboratory (SCDL), is defined for the purpose of performing operations and support for the aggregate of Embedded Computer Systems (ECS) assigned to Oklahoma City Air Logistics Center. The major objectives of SCDL are to:

- Establish a capability for software change implementation and module/ CPCI verification testing.
- Provide a basic framework for a multi-system support tool which encourages standardization and consolidates resources.
- Include within the framework a multi-purpose emulation tool which supports training and allows for exhaustive diagnostic probes.

The software change process during the Operations and Support phase of the ECS life cycle is described, and the range of applicability indicated for the subject tools. A set of requirements is stated for the tools as they are used for the individual ECS's as well as across the aggregate of ECS's. A configuration for the tool which satisfies the requirements is next selected which features a simulation processor complex with a set of common processors which functions as the simulation host processors for each of the Software Test Stands (Fig. 1). These Software Test Stands consist of the actual or emulated avionic processor loaded with the operational flight programs during testing.

Within this configuration framework, weapon system software support needs are examined for the B-52 Weapon System (Fig. 2), Generalized Software Test Stand (Fig. 3), and the Short Range Attack Missile (Fig. 4). Based on this analysis a baseline philosophy is established for the simulation processor complex (Fig. 5), and a proposal is made for an initial configuration of the SCDL (Fig. 6) to be located at Oklahoma City Air Logistics Center for use in Operations and Support.

A summary of the important points include: (a) analysis of a set of Weapon System ECS reveals where commonality requirements can be applied, (b) centralized simulation processor complex maximizes standardization while promoting modularity, (c) diagnostic emulation adds multi-system flexibility and supports training prior to system transfer, and (d) opportunity exists for applying the integration concepts across systems <u>before</u> individual facilities are established.

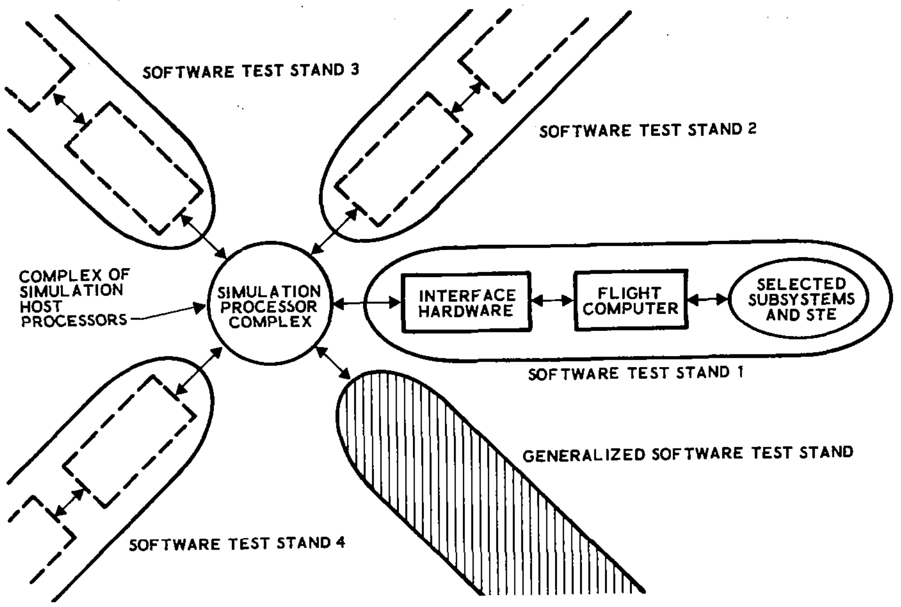

SOFTWARE TEST STAND 3
SOFTWARE TEST STAND 2
COMPLEX OF SIMULATION HOST PROCESSORS
SIMULATION PROCESSOR COMPLEX
INTERFACE HARDWARE
FLIGHT COMPUTER
SELECTED SUBSYSTEMS AND STE
SOFTWARE TEST STAND 1
GENERALIZED SOFTWARE TEST STAND
SOFTWARE TEST STAND 4

Figure 1

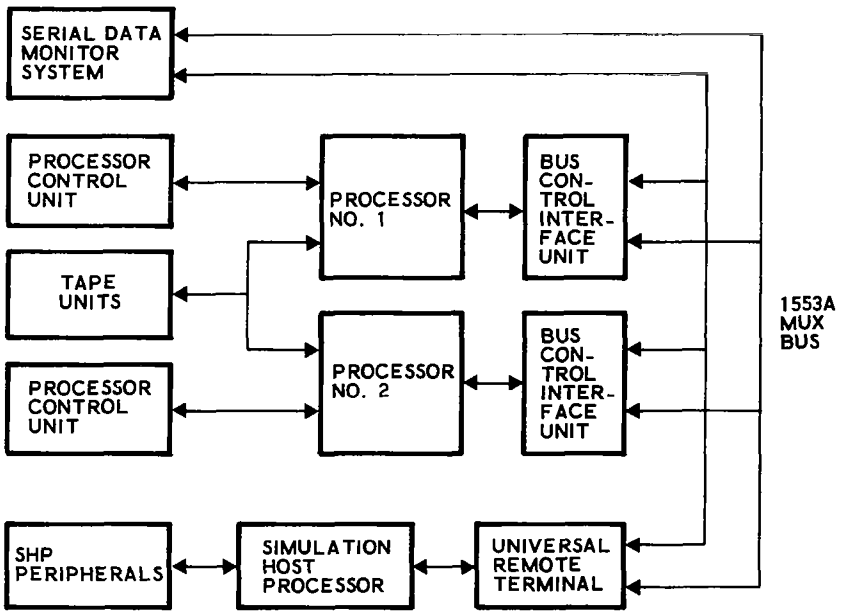

SERIAL DATA MONITOR SYSTEM
PROCESSOR CONTROL UNIT
PROCESSOR NO. 1
BUS CON-TROL INTER-FACE UNIT
TAPE UNITS
1553A MUX BUS
PROCESSOR NO. 2
BUS CON-TROL INTER-FACE UNIT
PROCESSOR CONTROL UNIT
SHP PERIPHERALS
SIMULATION HOST PROCESSOR
UNIVERSAL REMOTE TERMINAL

Figure 2

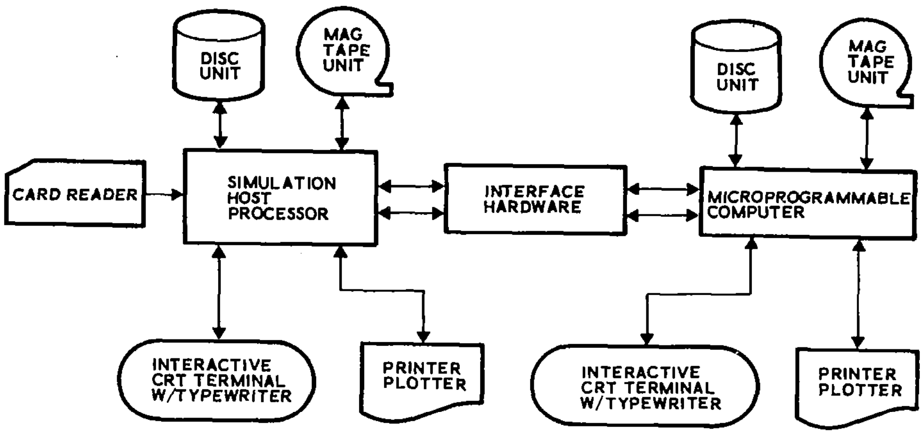

DISC UNIT
MAG TAPE UNIT
CARD READER
SIMULATION HOST PROCESSOR
INTERFACE HARDWARE
DISC UNIT
MAG TAPE UNIT
MICROPROGRAMMABLE COMPUTER
INTERACTIVE CRT TERMINAL W/TYPEWRITER
PRINTER PLOTTER
INTERACTIVE CRT TERMINAL W/TYPEWRITER
PRINTER PLOTTER

Figure 3

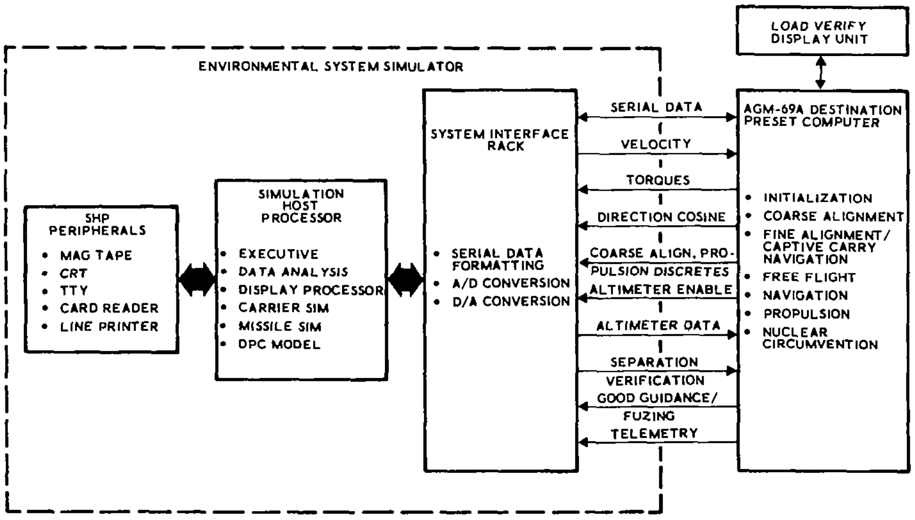


Figure 4

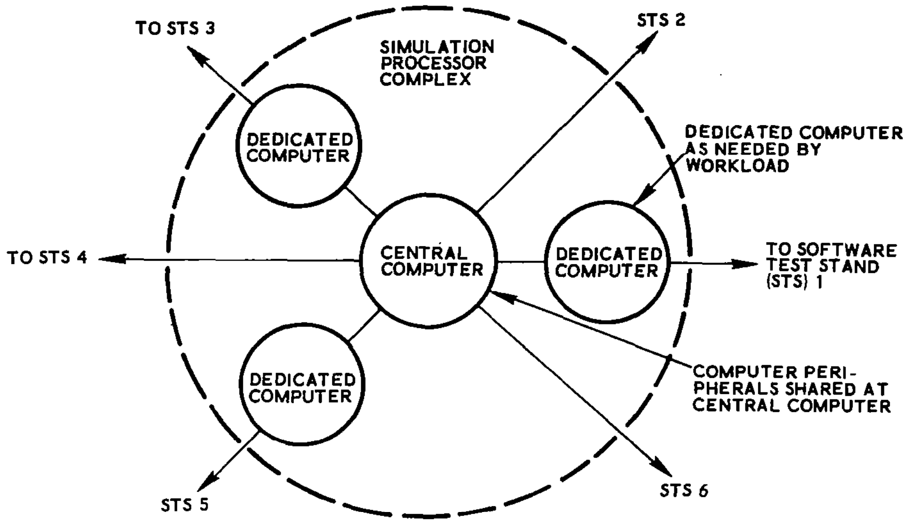

TO STS 3
SIMULATION PROCESSOR COMPLEX
STS 2
DEDICATED COMPUTER
DEDICATED COMPUTER AS NEEDED BY WORKLOAD
TO STS 4
CENTRAL COMPUTER
DEDICATED COMPUTER
TO SOFTWARE TEST STAND (STS) 1
DEDICATED COMPUTER
COMPUTER PERIPHERALS SHARED AT CENTRAL COMPUTER
STS 5
STS 6

Figure 5

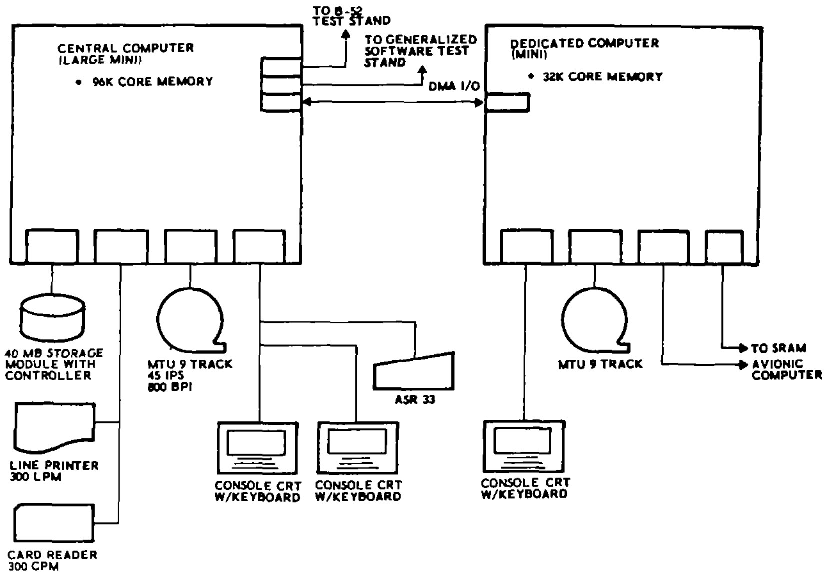


Figure 6

TOOLS AND SOFTWARE REQUIREMENTS, DESIGN, AND SPECIFICATIONS

Lt. Col. Charles John Grewe, Jr.
U.S. Air Force Electronic Systems Division

THE PROCESS

I am almost sure that everyone is familiar with the concept that you can divide the software development process into distinguishable phases. At least one way of looking at this division is to consider the phases as Design, Code, and Test. An estimate of the amount or percent of the total development effort is that the Design Phase requires about 45%, Coding requires about 20%, and Test requires the remaining 35% of the effort associated with the software development process. Figure 1 depicts the areas in which errors in the development process occur. As you can see, most of the errors occur in the Design Phase or can be attributed to poor, incorrect, or incomplete design when they are finally discovered. Of course, as you would expect, errors also occur in the Coding Phase. Unfortunately, however, errors are generally not discovered until the Test Phase. The Design Phase in this context does not include the area of Requirements Development, which I consider an entire Phase unto itself. I will elaborate on this concept later, but first, a few more words on the error discovery process and the impact it has on the cost of correction and effectively on the total cost of the software development project.

THE COST OF ERROR CORRECTION

Error correction is of itself an entire professional category. When you think of each and every profession in the environment today, a large portion of the cost and time associated with keeping devices, things, etc. going is associated with error correction. The military aircraft industry is a good example. How many times do we read about or hear about recalls necessary to correct an error in design, material, or workmanship. Millions of dollars are spent correcting errors discovered in systems (the airplane is also a system), and a large part, in fact a majority, of the cost is associated with correcting errors of design. Even worse, are errors associated with the market place (i.e., the needs or requirements) for the system. The curve shown in Figure 2 represents the associated higher costs of discovering errors as a function of the passage of time. Once a product is in production and you discover a design error (or, heaven forbid, find that no one wants the product), it is exponentially more expensive to correct or compensate for the mistake. This tool is true relative to the software development process. Mistakes are made in developing the requirements for a software system, in designing the software system, in coding the software system, and even in the testing (or lack of testing) the software system. So what should and can be done about it?

REQUIREMENTS DEVELOPMENT

I have referred to Requirements Development as a Phase in the software development process. This is a generous phase that allows one to talk about all of the activities associated with trying to understand what the needs and requirements are for the system or subsystem being developed. It covers such considerations as what the needs are, who needs them, how do various people perceive the needs, why are they needed, how do you write down or otherwise document the requirements for the needs, how do you analyze the requirements descriptions to be sure the needs (real or as perceived) were adequately documented, and how do you provide for tracing the documented needs to the provider of it. I include, as shown in Figure 3, under this general Phase the more commonly accepted terms or processes as:

-- Requirements Definition (Needs Determination)
-- Requirements Specification
-- Requirements Analysis
-- Requirements Traceability

The understanding of these terms is as varied as there are other ways of categorizing the different processes of Requirements Development. There is probably little hope in considering that a standard could be agreed on and adhered to in our industry of weapon system and software subsystem development and acquisition. Therefore, descriptions of these phrases are short in order to leave room for interpretation.

Requirements Definition is the process of understanding what the needs of all interested parties are and documenting these needs as written definitions and descriptors.

Requirements Specification is the more formal process of taking the descriptions of the needs and associating them with a formal structure (i.e., the format of MIL-STD-490 Type A System Level Specifications), such that they are presentable to someone else for implementing.

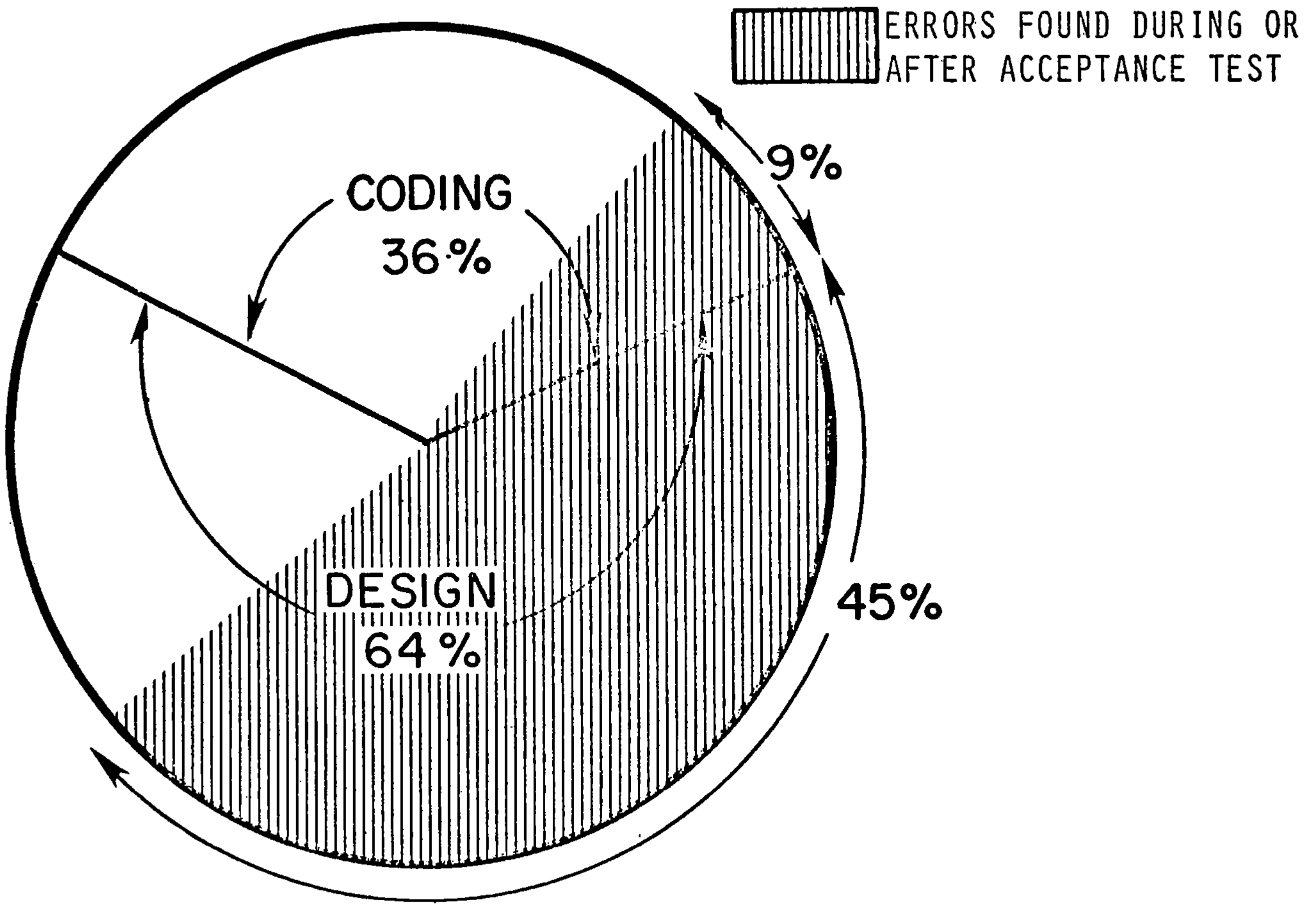
Requirements Analysis is the process of applying various scenarios to the statements of system requirements to assist in determining if the specifications are complete, consistent, feasible, have alternative possible solutions, and are understandable.

Requirements Traceability is the process of documenting where the stated requirements come from and how they are associated with, or relate to, other requirements.

Of the papers being presented here today, the first two by Melliar-Smith of SRI International and E. Rang of Honeywell, Inc. address aspects of software design and verification through the use of formal mathematical methods and the possible application of three other software development aids that make it "easy" to verify aspects of the system design. The next two papers by Paul Scheffer of Martin Marietta Aerospace and Judith Enos of Hughes Aircraft Company address the broader scope of software development to incorporate aspects of requirements development, system design, and software development support. The fifth paper, by W. Riddle of the University of Colorado, presents the DREAM software design aid toolbox and identifies an approach that involves the identification of operational events, constraints on the occurrence of events, and provides for the development of a system structure to support the event/constraint situation. The final paper, by J. Wileden of the University of Massachusetts, discusses a technique for use in describing dynamically structured, concurrent software systems.

These papers each address one or more aspects of Software Requirements, Specification, and Design Tools and should provide you with further insight into the capabilities that exist today and possibilities for the future.

SOFTWARE ERROR SOURCES (CCIP-85 DATA: 220 ERROR TYPES)



- LATE DISCOVERY OF AN ERROR HAS LARGE COST AND SCHEDULE IMPACT

Figure 1

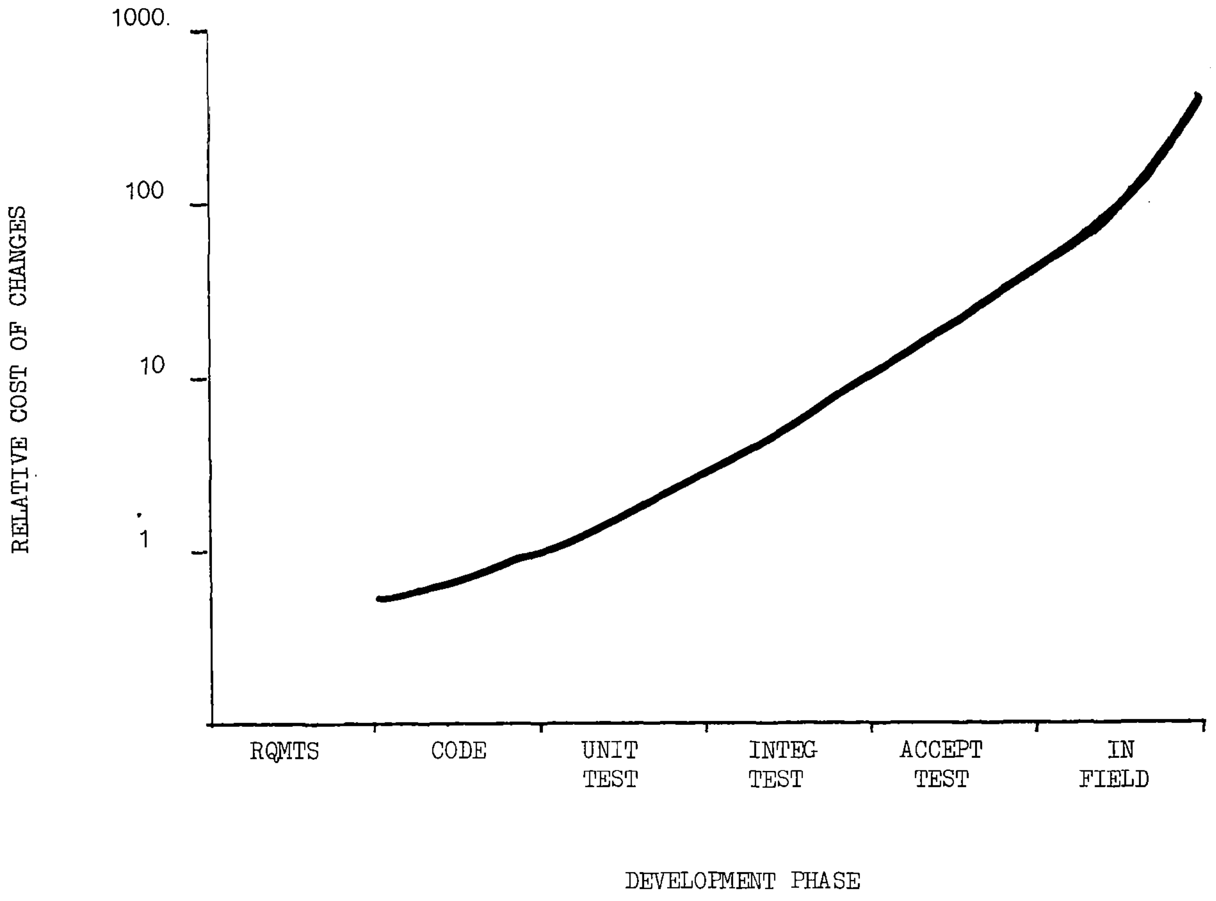

1000.
100
10
1
RELATIVE COST OF CHANGES
RQMTS
CODE
UNIT TEST
INTEG TEST
ACCEPT TEST
IN FIELD
DEVELOPMENT PHASE

Figure 2

REQUIREMENTS DEVELOPMENT

- REQUIREMENTS DEFINITION
- REQUIREMENTS SPECIFICATION
- REQUIREMENTS ANALYSIS
- REQUIREMENTS TRACEABILITY

Figure 3

THE DEVELOPMENT OF PROGRAMS TO MEET EXCEPTIONAL RELIABILITY REQUIREMENTS *

P. M. Melliar-Smith
SRI International

As modern aircraft flight control systems become more complex and critical to aircraft safety, so the certification of the systems becomes increasingly difficult. These flight control systems demand an exceptional level of reliability, beyond that required of any other computer application and well beyond what can be assured, or demonstrated, by conventional methods. Moreover the increasing complexity of the systems makes the attainment, and particularly the confirmation, of this reliability increasingly difficult.

The traditional methods of program development and testing can produce programs whose reliability suffices for many purposes. The reliability required of such programs is low enough that it is possible to measure the failure rate during testing or service and to use these measurements to estimate the reliability of the program. However the reliability required of safety-critical flight control programs is such that even many years of testing or service in many aircraft will not suffice to provide the data confirming that level of reliability. Thus we can no longer depend on the the traditional approaches, which locate and remove faults individually until the required reliability is attained. Only approaches that can guarantee in advance that no faults remain will be able to convince us that the flight control system will meet the exceptional reliability requirement.

It is interesting to contrast the capabilities and limitations of the alternative methods of assuring the reliability of programs. Thus flight testing must necessarily be only a partial test, which cannot find all the possible faults but which makes very few assumptions about the system. In contrast, program proof is extremely thorough and ensures the absence of faults but only within the specific assumptions made about the system. A part of the value of flight testing is that it can be used to confirm those assumptions. Testing in an aircraft simulator, which can afford a much more extensive but still inadequate test sequence, makes fewer assumptions about the system. Designs based on software fault tolerance use a different set of assumptions, and have the advantage that they are effective against faults that are not detected until the system is in service.

Recent advances in the application of formal mathematical methods to computer programs have influenced program design, testing, proof, and fault tolerance. In particular, the development of formal specifications for programs now provides a basis point for all of these activities. Further the semantic analysis of programs, developed for proof, is starting to influence the testing of programs and also the acceptance tests of the fault tolerance techniques.

*This research is supported in part by contracts from the Flight Electronics Division, NASA Langley Research Center.

EXCEPTIONAL RELIABILITY REQUIREMENTS

Safety Critical Computer Systems for Flight Control are allowed 10^-10 failures per hour, for the hardware and programs together. Most of this allowance must be allocated to the hardware.

Reliability of this order cannot be confirmed (but can be refuted) by testing, or even by extensive experience of use.

Certification of Safety Critical Computer Systems will therefore require novel methods of increasing our confidence in the programs.

Several novel methods are being developed, each with advantages and disadvantages. No one method can provide, at a reasonable cost, the degree of confidence necessary for certification.

The greatest degree of confidence in the computer system, and the most economical approach to certification, will come from an appropriate combination of all of these methods.

Figure 1

ALTERNATIVE APPROACHES TO RELIABLE PROGRAMS

FORMAL DESIGN METHODS	- essential, both for fewer program faults and for feasibility of systematic verification.
MANUAL INSPECTION	- inexpensive but more effective than testing, good for design faults, less good for detail.
TESTING IN SIMULATOR	- fault coverage too low for high reliability, depends on accuracy of assumptions in model.
FLIGHT TESTING	- very expensive but essential for credibility, to confirm assumptions - not to find faults.
PROGRAM PROOF	- expensive but extremely thorough, depends on accuracy of assumptions in model,
FAULT TOLERANT DESIGN	- effective against faults remaining in system, complementary to other approaches.

Figure 2

DESIGN METHODOLOGY AND FORMAL SPECIFICATION

FORMAL SPECIFICATION -	if we do not know what the system is to do, how are we to say whether it did it at all let alone reliably?
	the design,) the testing,) the proof,) the fault tolerance) all should be driven by formal specification.
FORMAL DESIGN METHOD -	to provide a structure that breaks a complex problem into several simpler problems,
	to provide levels of abstraction which allow us to think more easily about the problem,
	to allow us to talk about important aspects of a design without trivial obscuring details.

Figure 3

TESTING

Used to get a 'working' program rather than a very reliable program.

Simple test sequences can find many faults, but do not have fault coverage sufficient to find essentially all faults. The needed reliability can be obtained only by finding essentially all faults.

Systematic methods have been developed for generating exhaustive test sequences that can find all faults in a program. These test sequences are very large and impossible to use in practice.

To reduce the length of the test sequences, mathematical analysis of the program logic can be used. The analysis required to reduce the test sequence to a reasonable size is equivalent to program proving.

The value of testing is not to find faults or to ensure the absence of faults, but rather to confirm the validity of assumptions made by the other methods of ensuring reliability.

Figure 4

PROGRAM PROOF

Now becoming feasible for specially designed programs, but still very expensive.

Almost all experience is for programs that do logical manipulation. No experience yet for linear control systems (let alone nonlinear).

The method depends on a very detailed formal model of the computer, the sensors, the actuators, the aircraft dynamics, etc.
The proof is only as valid as this model and its assumptions.

The method proves that the program satisfies a formal specification.
The proof is only as valid as that formal specification.

Within these assumptions, program proof totally precludes any faults.

Timescale to completed demonstration for flight control: 3 - 5 years,
to first certification based on program proof: 10 years.

Figure 5

FAULT TOLERANT DESIGN

Error Detection and Error Recovery sections of programs are liable to be much less reliable than ordinary program. Special case methods are too complex to provide sufficient reliability.

The favored approach is that of Recovery Blocks.

Recovery Blocks require: Acceptance Tests,
Alternative Program Blocks,
State Restoration Mechanism.

Acceptance Tests, derived from the Formal Specifications, present most of the difficulties, and cause most of the overhead and cost.

Recovery Blocks cannot provide the certainty of proof, but they make fewer assumptions about the system. Thus they provide a useful complement to proof in increasing our confidence in the system.

Figure 6

DESCRIBING A TRIPLY-REDUNDANT FLIGHT CONTROL SYSTEM FOR VERIFICATION OF DESIGN

Edward R. Rang
Honeywell, Inc.

INTRODUCTION

Remedies for many of the ills of software developments are known and are explained in text books [1,2]*. At Honeywell, our design guidelines are in the second edition [3,4]. Yet we still have some problems in putting the remedies to practice. For flight control systems part of this is due to the nature of the system and part may be due to the manner in which we do business. I would like to talk about these things in the context of describing the first level of the software design for a triply-redundant system. Our primary tool for descriptions is HIPO [5]. We have investigated the SRI International techniques [6,7,8] and a methodology based on Dijkstra's constructive approach [9]. I will make some subjective comments on these.

FLIGHT CONTROL SOFTWARE

The programs are of moderate size, less than 10 000 lines of assembly code. The functions separate nicely and allow the program to be well structured with easily defined interfaces between modules. The control and data structures are elementary. There are only a few do-while loops for synchronization and shut-down. For these, the termination is obvious. Hence, symbolic evaluation will yield a formal verification.

Most of the functions can be constructed as finite-state automata. The outputs are computed in terms of the inputs and the current state values. Then the transition is made to the next set of state values. Holding to this makes the design of the mode logic functions, the signal select mechanisms and the failure management facilities very certain.

*Number in brackets indicates reference.

The basic simplicity of flight controls has permitted the engineer to provide acceptable software designs without using structured programming or other disciplines. The documentation of the software and the description of the design has often been far from adequate.

HIPO DESCRIPTIONS

With proper diligence, a hierarch-plus-input-output presentation may be made complete and sufficiently rigorous to verify the flight control system design. Attention must be paid to supplying the motivations, a list of the functional capabilities which are required or other descriptions of the purpose of the module. The variables which carry state information should be specified. But there is nothing in the HIPO charting to enforce this. One can write complete and accurate HIPO charts that are as stark as assembly code.

The processes for synchronization in a triply-redundant system are not complicated. Formal descriptions are possible using Petri nets [10]. We have also studied synchronizations with simulations.

SRI INTERNATIONAL METHODOLOGY

This approach adds a formal discipline at the specification level. It provides checks for syntax, checks for the consistency of references between modules, and checks for circular references between modules. It provides a discipline to ensure that the state variables are specified and initialized. The tools are well designed and easy to use. However, there is vastly more generality and capability in their methodology than is required for our modest flight control system. The separation of functional capability, described by the non-procedural language SPECIAL, and the sequencing of functions in time, described by their program language, is a level of sophistication that obscures our simple programs. It is just as difficult to add motivational comments and general descriptive material to SPECIAL listings as it is to add that material to assembly code. It is more natural on the HIPO charts.

A CONSTRUCTIVE METHODOLOGY

The approach based on Dijkstra's work [9] aims at supplying verification along with the design. Attention is focused on loop invariants for computations of very clever algorithms. This does not help our problem much. The general design and documentation tools are useful in the large-scale software development cycles, but flight controls can manage with much less.

CONCLUDING REMARKS

My contention is that the design of a triply-redundant flight control system may be described using HIPO charts in sufficient detail and with sufficient rigor to verify by oral demonstration that the system performs all of its intended functions and does not perform any unintended functions. This is facilitated by constructing the functions as finite-state automata. The formal methodology constructed by SRI Internation is helpful but not absolutely necessary since the flight control software is not a particularly difficult problem.

REFERENCES

1. Tausworthe, R. C.: Standardized Development of Computer Software. Prentice-Hall, 1977.

2. Yourdon, Edward: Techniques of Program Structure and Design. Prentice-Hall, 1975.

3. Bailey, D. G.: Airborne Software Structured Development. Rep. ED-DGM 4000-310 (A), Honeywell Avionics Div., Feb. 1978.

4. Lane, David: Structured Programming Guidelines. Rep. 776-12882, Vols. I & II, Honeywell Avionics Div., Sept. 1977.

5. Katzan, Harry, Jr.: Systems Design and Documentation - An Introduction to the HIPO Method. Van Nostrand Reinhold, 1976.

6. Robinson, L.; and Levitt, K. N.: Proof Techniques for Hierarchically Structured Programs. Comm. ACM, vol. 20, no. 4, Apr. 1977, pp. 271-283.

7. Roubine, O.; and Robinson, L.: SPECIAL Reference Manual. Third Ed. Tech. Rep. CSG-45, Standford Res. Inst., Jan. 1977.

8. Boebert, W. E.; Kamrad, J. M.; and Rang, E. R.: The Analytic Verification of Flight Software: A Case Study. NAECON 1978, Vol. 1, May 1978, pp. 242-248.

9. Boyd, D. L.; Pizzarello, A.; and Vestal, S. C.: Rational Design Methodology. Honeywell Rep. to RADC, HR-78-257: 17-38, June 1978.

10. Peterson, J. L.: Petri Nets. Computing Surveys, vol. 9, no. 3, Sept. 1977, pp. 223-252.

SOFTWARE DESIGNERS WORKBENCH

Requirements and Design Tools for Expression and Evaluation

Paul A. Scheffer
Martin Marietta Aerospace

One of the more innovative of recent software engineering activities is the concept of a Programmer's Workbench (PWB). The PWB is a very different approach to improving the software development process. It is based on a program development "facility" much like those that have been developed for other professions (e.g., carpenter's workbench, engineer's laboratory). This approach helps focus attention on the need for adequate tools and procedures; it serves as a mechanism for integrating tools into a coordinated set; and it tends to add stability to the programming environment by separating the tools from the product. The PWB idea separates the Workbench, which performs the development and maintenance function, and the host or target computer on which the production system will run. The link between the two machines represents a physical connection which is used to transfer data, run tests, etc.

A simple generalization of the PWB concept results in the idea of a total software engineering facility, i.e., a Designer's Workbench (DWB). In the DWB concept, the library of PWB support tools to build and manipulate program source code is expanded to handle the complete set of functions needed for all software design and development activities. Such functions include requirements and design component identification, specification, documentation, management control support, design evaluation mechanisms, and so on.

At Martin Marietta/Denver, we are actively pursuing the development of this concept. In addition to installing a software development laboratory as a dedicated facility, we have several plans for expanding the tools repertoire available. Using a PDP 11/70 architecture base, and system software which includes RSX-11M and the UNIX operating systems, standard PWB tools, and INGRES data base capabilities, we are augmenting the system with several high level DWB tools. These design tools basically take two forms. One is a set of languages for expressing design: the set consists of a language for each design level (currently expected to be 3) starting with requirements definition. The second form of design tool is closely associated with the languages and satisfies a design evaluation function. Evaluators provide feedback information to the designer on what has been expressed in the language. The evaluation can be an assessment of what has been syntactically stated in the language (consistency reports, summaries, etc.), a semantic assessment that is static in nature (design structure), or a dynamic (simulative) assessment.

This presentation considers only those parts of this overall software engineering facility plan which have been implemented and are being used. On the language side, this includes a high level requirements language called MEDL-R and an initial design phase language called MEDL-D. These two languages have been carefully defined to allow a smooth transition from one to the next. Primarily this is to support the traceability and management of individual system requirements. Design evaluation tools are discussed in terms of analytics for both requirements and design component structure, and in terms of quantification of design quality. The analytic tools for evaluation provide information on design in a manner similar to that provided by source code analyzers. Design quality measures will provide feedback on the degree to which a design satisfies various characteristics of quality such as maintainability, modularity, or testability.

<u>SOFTWARE DESIGNERS WORKBENCH</u>

General:
- o Support <u>Any</u> Phase of S/W Development
- o "Comfortable" User Interface
- o Extensible, Total System Development

Specific:	Requirements	Preliminary Design
	Definition	Structure
	Management	Data Definition
	Analysis	Interfaces
Future:	Detailed Design	Simulation
	Architecture	H/W - S/W Symbiosis
	Processes	Resources
		Behavior

Figure 1

MULTI-LEVEL EXPRESSION DESIGN LANGUAGE SYSTEM

- o Fills Void of Requirements Language (MEDL-R)
- o Specific Language for Each Design Phase, Leading to End-to-End Support System
- o Assessment Methods to Evaluate a Design at Any Level
- o Measurement Techniques Attempt to Quantify Characteristics of Quality

Figure 2

STATUS

- o MEDL-R: Implemented, Undergoing V & V Using NASA "Live" Requirements Test-Bed
- o MEDL-D: Still in Design Activity
- o CSEF: Operational, Anticipated Hardware Expansion
- o Design Languages: Surveys and Evaluations On-Going, SSL Model for MEDL-D
- o RISS/MASS: Implemented Relational DBMS for CSEF

Figure 3

MEDL ASSESSMENT

- o Requirements Level Unique in Supporting True Requirements Data Base
- o Information "Explosion" a Potential Problem
- o Reduces Manageability Problems, especially for Volatile Requirements
- o Serves Throughout Life-Cycle of S/W Development
- o Crossover, MEDL-R to MEDL-D to MEDL-P, Satisfies Traceability Needs
- o Inexpensive Tool (11/70); Requires DBMS Currently Built in FORTRAN

Figure 4

CONCLUSIONS

- o DWB Concept - Properly Focuses Emphasis on Tools
- o Cost-Effectiveness - Benefits with just a Few Tools (PWB) Seem to Justify Further R & D and Library Expansion
- o S/W Design Model - All-Inclusive Model (Concept Formation to Maintenance) is Lacking
- o Design Evaluation Mechanisms - Need Further Research for Calibration, especially Quality Aspects

Figure 5

TOOLS FOR EMBEDDED SOFTWARE DESIGN VERIFICATION

J. C. Enos and R. R. Willis
Hughes Aircraft Company

SUMMARY

One of the many ways in which Hughes is attacking the software life cycle cost problem is by engineering tools which verify that designs meet intended requirements. These tools support the user by providing automatic feedback for assessing the consistency, completeness, performance, and quality (cost) of software designs. The tools and their application to known problems are summarized in Table 1.

TOOL CAPABILITIES

Structured Design (ala Constantine/Meyers) is Hughes' methodology for developing the structure of software components. To support designers, we have developed the Structure Chart Graphics System (SCG) and the Design Quality Metrics System (DQM). SCG automates the development and formal documentation of structure charts using graphics terminals, plotter outputs, and a computer data base of the structure. DQM uses the data base to automatically quantify the extent to which structured design guidelines have been adhered to. I.e., DQM enables the designer to evaluate the cost or goodness of the design.

Hughes makes extensive use of representative simulation models to verify that designs meet required performance criteria. To support the analysts we have developed the General Function Model (GFM) and the Distributed Data Processing Model (DDPM). The GFM provides a general simulation model for evaluating operational feasibility. The DDPM provides a vehicle for quantifying design tradeoffs for architecture, allocation, hardware selection, and software design.

These tools have been integrated or are scheduled for integration into an automated support facility, the Design Analysis System (DAS). The DAS currently provides a user-engineered graphics interface, automatic model generation and interactive graphics support to the General Function Model for operational feasibility analysis (DAS/OFD). PSL/PSA (University of Michigan ISDOS project) supplements the system with automatic documentation, common data base maintenance, and consistency and completeness checking.

FUTURE PLANS

The DAS is an evolving system. All tools as described are currently available either as an integrated part of the system or as stand-alone tools. To address the technology transfer problem for Structured Design we are engineering a semantic preprocessor for the SCG System which will enforce, on a company wide basis, the standards established through use of the methodology on a number of projects. To address the requirements communication problem, we are engineering a requirements analysis system which will include: 1) a graphic system for construction, maintenance, and documentation of information flow diagrams, 2) simulation modelling aids such as QGERTS and GMB (UCLA SARA project), and 3) schema for mapping these data into PSL for automated traceability.

TABLE 1.- SOFTWARE DESIGN ANALYSIS TOOLS

PROBLEM	SOLUTION	TOOLS	STATUS
DELIVERED SYSTEMS DON'T MEET INTENDED REQUIREMENTS	1. RIGOROUS REQUIREMENTS ASSESSMENT VIA CUSTOMER INVOLVEMENT IN MODEL DEVELOPMENT 2. *RIGOROUS OPERATIONS* ANALYSIS VIA SIMULATION MODELS. 3. REQUIREMENTS ANALYSIS FOR CONSISTENCY AND COMPLETENESS	1. INFORMATION FLOW DIAGRAMMING 2. DESIGN ANALYSIS SYSTEM/ OPERATIONAL FLOW DIAGRAM *(DAS/OFD)—INCORPORATES* GENERAL FUNCTION MODEL (GFM) WITH GRAPHICS INPUT OF OPERATIONAL FLOW DIAGRAMS (OFDs) AND INTERACTIVE GRAPHICS AID TO SIMULATION ANALYSIS (GAS) 3. PSL/PSA (ISDOS)/CARA	1. IN USE ON 2 MAJOR CONTRACTS 2. DAS/OFD DEMONSTRATED ON INTERNAL RESEARCH AND DEVELOPMENT — GFM AND *PREDECESSOR MAN MACHINE* MODEL (MMM) USED ON 6 MAJOR PROJECTS — NOW STANDARD FOR ALL NEW PROJECTS 3. INTERNAL RESEARCH AND DEVELOPMENT INVESTIGATION *COMPLETE—TECHNOLOGY* TRANSFER VIA PILOT PROJECT SCHEDULED FOR 1/79
TRADE-OFF DESIGN DECISIONS, ESPECIALLY FOR COMPLEX DISTRIBUTED PROCESSING NETWORKS, DIFFICULT AND COSTLY	GENERALIZED SIMULATION MODELS TO SUPPORT DISTRIBUTED PROCESSING TRADE-OFF ANALYSIS	DISTRIBUTED DATA PROCESSING MODEL (DDPM)	DDPM,AND ITS PREDECESSOR DPM USED ON 10 MAJOR PROJECTS — NOW STANDARD FOR ALL NEW PROJECTS — INTEGRATION WITH DAS SCHEDULED FOR 1979
APPLICATION OF STRUCTURED DESIGN TECHNIQUES IS UNCONTROLLED AND NON-STANDARD. TECHNOLOGY TRANSFER TO PROJECTS IS SLOW	USER ENGINEERED INTERFACE TO INTERACTIVE TOOLS WHICH STANDARDIZE USE OF DESIGN METHODOLOGY	*STRUCTURE CHART GRAPHICS* SYSTEM (SCG) DESIGN QUALITY METRICS (DQM)	*IN USE ON JTIDS-SEMANTICS UP* GRADE REFLECTING STRUCTURED DESIGN METHODOLOGY USE ON 10 MAJOR PROJECTS IN WORK

THE DESIGN ANALYSIS SYSTEM

AN INTEGRATED APPROACH TO TOOL DEVELOPMENT

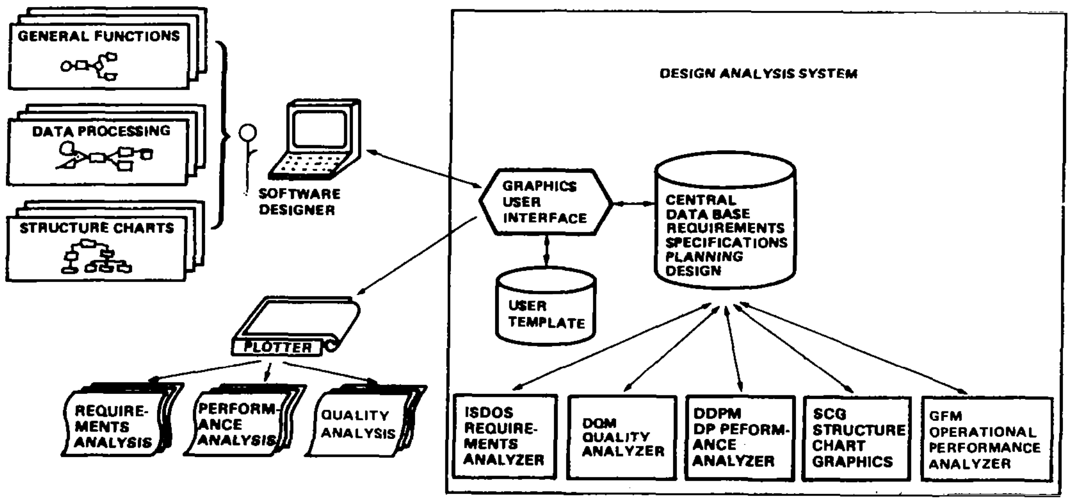


Figure 1

DAS/OFD

AN AID TO EVALUATION OF USER REQUIREMENTS FOR OPERATIONAL CONCEPT FEASIBILITY

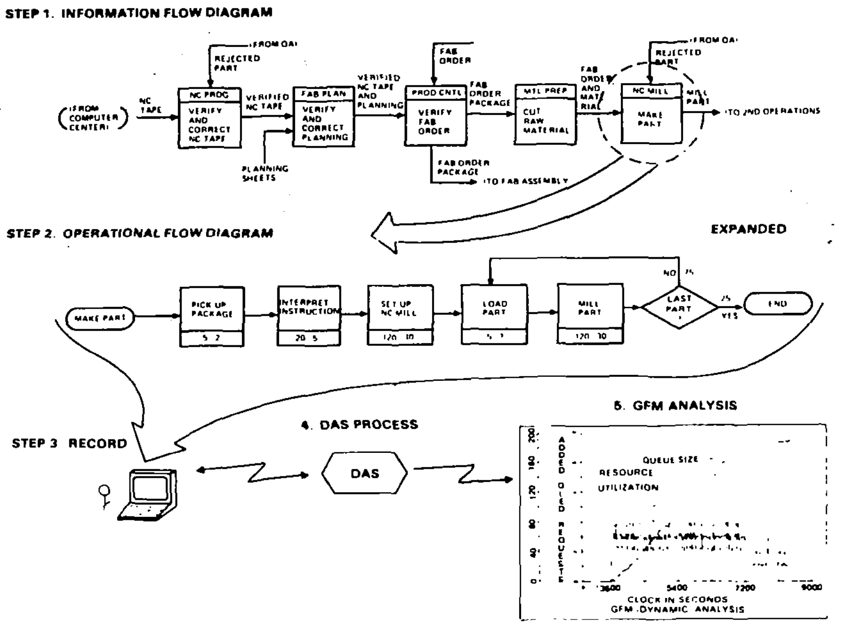
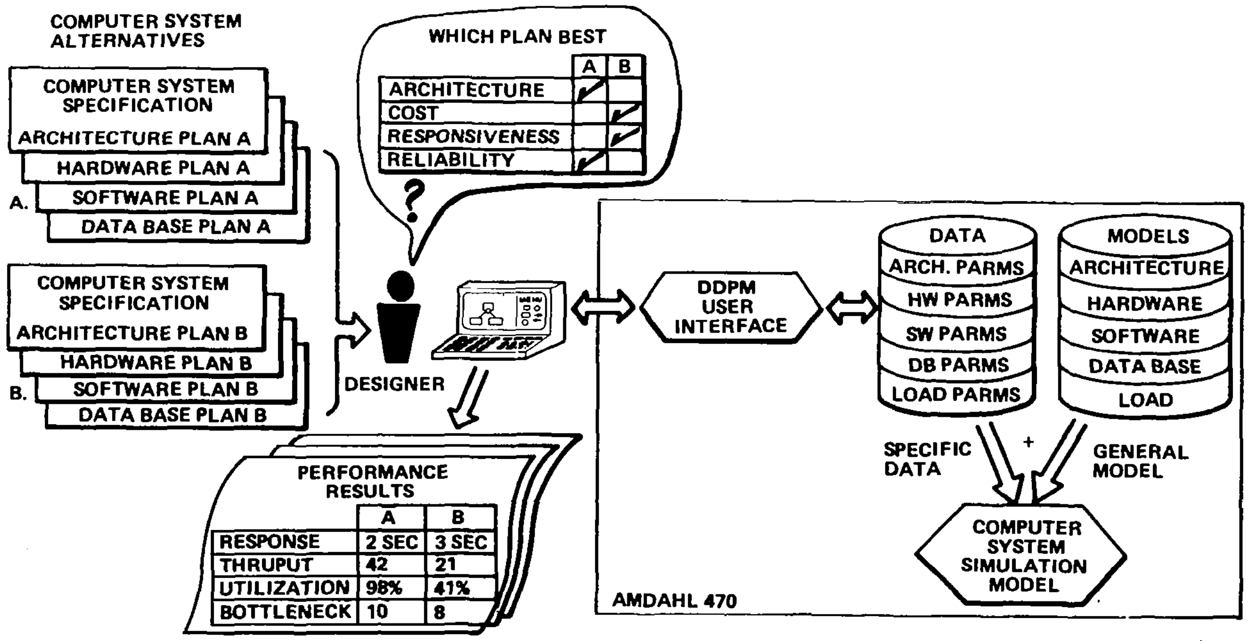


Figure 2

THE DDPM

A TABLE-DRIVEN MODEL FOR SIMULATION OF COMPUTER SYSTEM DESIGN ALTERNATIVES

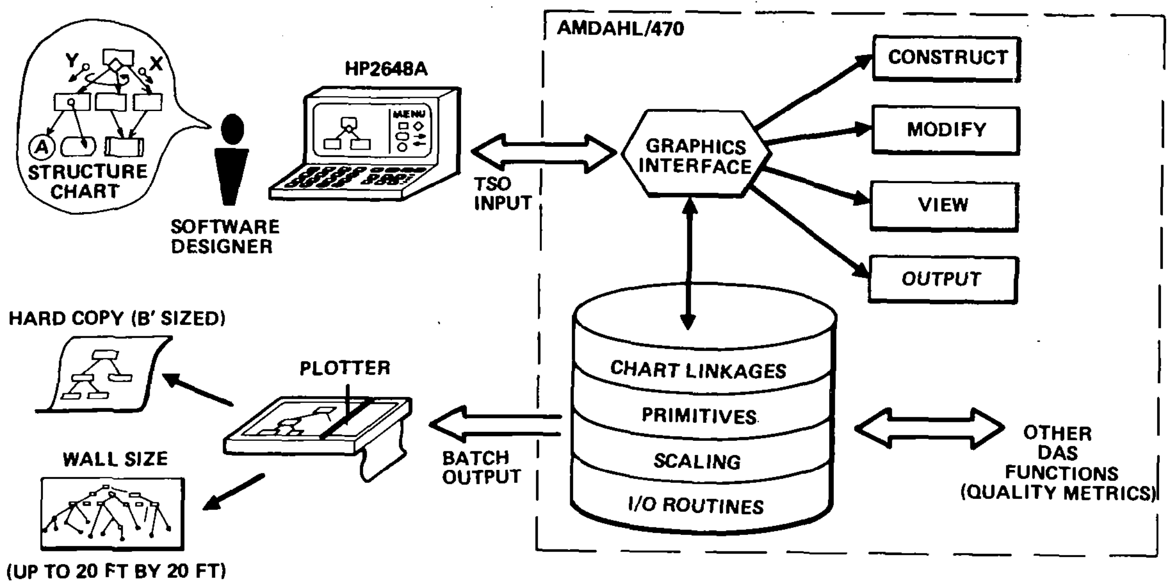


	A	B
RESPONSE	2 SEC	3 SEC
THRUPUT	42	21
UTILIZATION	98%	41%
BOTTLENECK	10	8

Figure 3

STRUCTURED DESIGN GRAPHICS SYSTEM

AN AID TO STRUCTURED DESIGN PROVIDING AUTO-DOCUMENTATION AND METHODOLOGY STANDARDIZATION

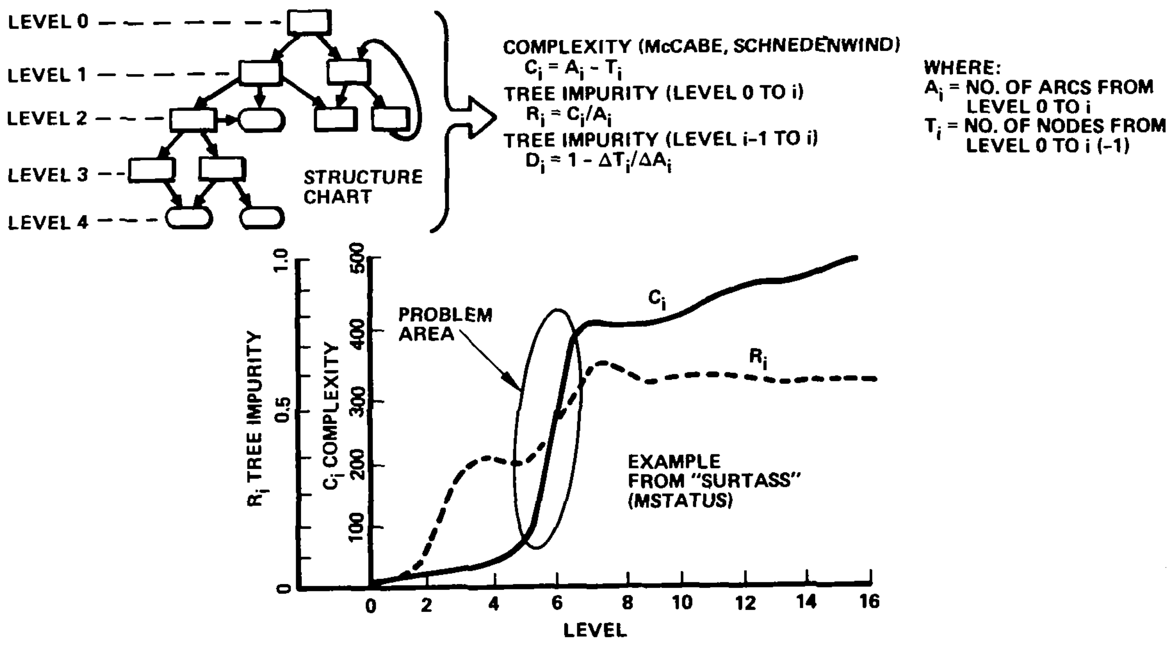


20 ft = 6.1 m

Figure 4

DESIGN QUALITY METRICS SYSTEM

A QUANTITATIVE MEASURE FOR DESIGN "GOODNESS"



COMPLEXITY (McCABE, SCHNEDENWIND)
$C_i = A_i - T_i$
TREE IMPURITY (LEVEL 0 TO i)
$R_i = C_i/A_i$
TREE IMPURITY (LEVEL i-1 TO i)
$D_i = 1 - \Delta T_i/\Delta A_i$

WHERE:
A_i = NO. OF ARCS FROM LEVEL 0 TO i
T_i = NO. OF NODES FROM LEVEL 0 TO i (-1)

Figure 5

AN EVENT-BASED DESIGN METHOD FOR EMBEDDED SOFTWARE SYSTEMS

William E. Riddle
University of Colorado at Boulder

The Design Realization, Evaluation And Modelling (DREAM) system has been developed to provide aid to the designers of complex software systems. Its major component is a language, called the DREAM Design Notation (DDN), which permits the rigorous description of a software system as it evolves during design. The language also permits the rigorous description of the environment in which the software system operates. The DREAM system supports a variety of design methods through tools which provide bookkeeping and decision-making aid to designers. A trial, partial implementation of the system has been completed and the design language has been used to describe a wide range of existing software systems. The language has also been used in several design experiments conducted to assess the effectiveness of the system during design, but more rigorous assessment of the system and language is needed.

In the course of the design experiments, however, a new variation of the traditional top-down, elaborative design method has been identified. In this event-based design method, the first step is to identify interesting events which occur during the operation of the system. Then constraints upon the occurrences of these events are rigorously defined. Then system components are demarcated and interactions among the components are defined which lead to the satisfaction of the constraints. These activities constitute a design step and produce a more complete, but still incomplete, design which is further elaborated by the next design step.

The event-based design method, as currently defined within the context of the DREAM system, is oriented toward the development of the processing strategies of software systems having components which operate concurrently. It introduces a desirable rigor into software system design and forces designers to consider global aspects of the system before developing the system's detail. The method also provides many opportunities for analysis, allowing designers to incrementally assess the appropriateness of their design decisions.

BIBLIOGRAPHY

Riddle, W. E.; Sayler, J. H.; Segal, A. R.; Stavely, A. M.; and Wileden, J. C. DREAM -- A Software Design Aid System. Proc. Third Jerusalem Conf. on Information Technology (Jerusalem), Aug. 1978.

Riddle, W. E.; Wileden, J. C.; Sayler, J. H.; Segal, A. R.; and Stavely, A. M. Behavior Modelling During Software Design. IEEE Trans. on Software Engineering, July 1978.

Riddle, W. E.; Sayler, J. H.; Segal, A. R.; Stavely, A. M.; and Wileden, J. C.: A Description Scheme To Aid the Design of Collections of Concurrent Processes. Proc. 1978 National Computer Conf. (Anaheim), June 1978.

THE DREAM SYSTEM

PURPOSE: provide aid to designers of complex software systems

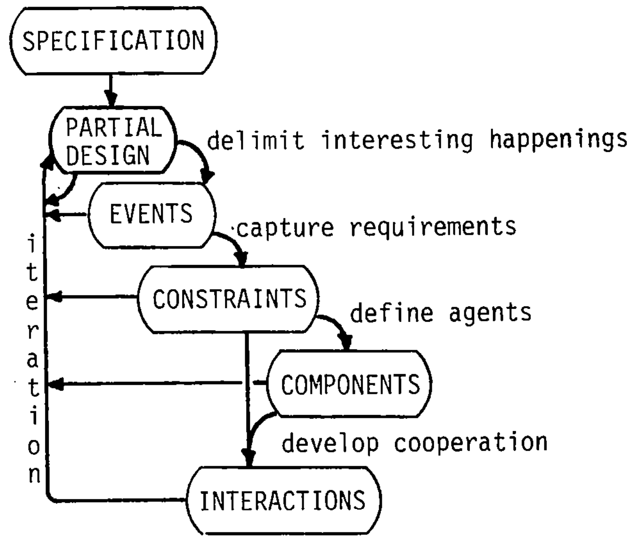
FACILITIES:
. DESIGN LANGUAGE for blueprinting the software and the environment in which it is embedded
. DESIGN METHODS for the gradual evolution of a system's architectural design
. DESIGN TOOLS to aid designers in the bookkeeping and decision-making tasks

STATUS:
. design language has been completely developed
. partial, trial implementation is complete
. several existing systems have been described:
    . operating systems
    . artificial intelligence systems
    . control systems
    . hardware systems
    . interactive computing systems
. a few design experiments have been conducted
. analysis techniques for the support of decision-making have been formulated

FUTURE:
. need to increase domain of systems which can be attacked:
    . dynamically structured systems
    . interrupt-driven systems
. need to expand the set of decision-making tools
. need a full, proof-of-concept implementation
. need to test existing design methods
. need to develop new design methods

Figure 1

EVENT-BASED DESIGN METHOD



COMMENTS:
. back-up may be necessary
. language allows decisions at each step to be recorded
. bookkeeping aid supports the preparation and modification of evolving description
. decision-making aid allows the completeness and consistency of the description to be checked at each step

Figure 2

EVENT DEFINITION

INTENT: delimit interesting happenings

EXAMPLE:
```
EVENT DEFINITION;
  heat_up: DESCRIPTION; temp ≥ 150°F END;
  handle_hot_engine:
    SEQUENCE (heat_up, notice, ring);
  END;
```

CONCEPTS USED:

- definition of primitive events
- definition of non-primitive events as sequences of other events:
  concatenation, repetition, alternation, concurrency, re-entrancy, synchronization

COMMENTS:

- a system is a collection of parallel parts
- the system is both the software portions and the other parts with which the software interacts
- start with an external view

Figure 3

CONSTRAINT DESCRIPTION

INTENT: capture requirements

EXAMPLE:
```
DESIRED BEHAVIOR;
  POSSIBLY 4 CONCURRENT
    (SEQUENCE (heat up, notice, ring));
  END;
```

CONCEPTS USED:

- requirements can be described in terms of desirable (or undesirable) event sequences

COMMENTS:

- description is non-prescriptive, admitting many ways to achieve the desired behavior
- description is non-procedural, specifying what (effect) rather than how (cause)
- the requirements that are captured may come either from the original specification or from considering incomplete parts of the design

Figure 4

COMPONENT DELINEATION

INTENT: define agents

```
EXAMPLE:  SUBCOMPONENTS;
            engines ARRAY [1..4] OF [engine];
            monitor OF [engine_monitor];
            alarm OF [audio_device];
            END;
```

CONCEPTS USED:

- systems decomposed hierarchically
- parts viewed as operating concurrently

COMMENTS:

- parts may be only hypothetical ones
- parts may be software or "hardware"

Figure 5

SUMMARY COMMENTS

Useful in defining the processing strategies employed in the system.

Useful for systems that decompose into parallel parts.

The method forces consideration of global aspects before consideration of detail.

The method leads to a desirable rigor in recording decisions at each step.

Useful analysis can be performed upon the evolving description; allows designers to incrementally assess the appropriateness of the design.

Figure 6

INTERACTION DEFINITION

INTENT: develop cooperation needed among components in order that constraints are observed

```
EXAMPLE:  [engine_monitor]: SUBSYSTEM CLASS;
            receive_status: IN PORT; END;
            sound_alarm: OUT PORT; END;
            observe: CONTROL PROCESS; MODEL;
              ITERATE RECEIVE receive_status;
                        MAYBE SEND sound_alarm;
                              END;
                        END;
              END; END;
          END;
```

CONCEPTS USED:

- interactions defined by message transfer
- pseudo-procedural definition of message handling

COMMENTS:

- redundant description allows consistency checking
- message transfer may or may not be the real mode of interaction

Figure 7

A TECHNIQUE TO AID IN THE DESIGN AND ANALYSIS OF DYNAMICALLY STRUCTURED, CONCURRENT SOFTWARE SYSTEMS

Jack C. Wileden
University of Massachusetts

Many contemporary complex software systems are most naturally described as collections of interacting parallel processes in which processes are created and destroyed or patterns of potential process interaction are altered during system execution. A description of this kind may be merely an accurate reflection of dynamic restructuring capabilities designed into the software system, as in the case of the RC4000 [1]* and HYDRA [2] operating systems or the HEARSAY [3] speech understanding system. Alternatively, this descriptive approach may be useful in explicitly representing potential modifications to a system's configuration which might result from the failure of processing elements or communications channels due to faults in either hardware or software. Thus, since the design of a complex software system should ideally be represented in the most natural possible manner, it seems evident that software system design tools should incorporate constructs for describing dynamically-structured parallel systems. As currently specified, however, design tools such as DREAM [4], SARA [5] and SREM [6] all base their system descriptions on a fixed set of processes and a fixed pattern of process intercommunication, with no provision for the natural representation of dynamic structure in a software system's design.

In this presentation, we discuss a technique developed for use in describing dynamically structured, concurrent software systems [7]. This technique is based upon a description language, called DYMOL, and an underlying formal model which provides a well-defined semantics for DYMOL'S constructs. Our technique, slated for future inclusion in the DREAM design aid system, is currently being employed as a rudimentary, stand-alone design tool and used in an investigation of cooperative distributed processing systems [8]. Our current assessment of the technique's potential utility as an aid to software system designers is largely based upon examples such as those described in the presentation. The examples considered to date suggest that, by providing capabilities for analyzing an evolving design, the technique can be of significant value to designers of dynamically structured, concurrent software. A more complete assessment of the technique's value must, however, await the development of automated versions which can serve as a basis for continued investigation and experimentation.

*Number in brackets refers to reference.

REFERENCES

1. Brinch Hansen, P.: The Nucleus of a Multiprogramming System. Comm. ACM, vol. 13, no. 4, Apr. 1970, pp. 238-241, 250.

2. Wulf, W.; Levin, R.; and Pierson, C.: Overview of the HYDRA Operating System Development. Proc. 5th Symp. on Operating Systems Principles, Operating Syst. Review (ACM SIGOPS Newsletter), vol. 9, no. 5, Nov. 1975, pp. 122-131.

3. Fennel, R. D.; and Lesser, V. R.: Parallelism in AI Problem Solving: A Case Study of HEARSAY-II. IEEE Trans. Comp., vol. C-26, no. 2, Feb. 1977, pp. 98-111.

4. Riddle, W. E.; Wileden, J. C.; Sayler, J. H.; Segal, A. R.; and Stavely, A. M.: Behavior Modelling During Software Design. IEEE Trans. Software Eng., vol. SE-4, no. 4, July 1978.

5. Estrin, G.; and Campos, I.: Concurrent Software System Design, Supported by SARA at the Age of One. Proc. 3rd Int. Conf. on Software Eng., 1978, pp. 230-242.

6. Davis, C. G.; and Vick, C. R.: The Software Development System. IEEE Trans. Software Eng., vol. SE-3, no. 1, Jan. 1977, pp. 69-84.

7. Wileden, J. C.: Modelling Parallel Systems With Dynamic Structure. COINS Tech. Rep. 78-4, Dep. Computer & Information Sci., Univ. of Massachusetts, Jan. 1978.

8. Lesser, V. R.; and Corkill, D.: Cooperative Distributed Processing: A New Approach for Structuring Distributed Systems. COINS Tech. Rep. 78-7, Dep. Computer & Information Sci., Univ. of Massachusetts, 1978.

PARALLEL SYSTEM WITH DYNAMIC STRUCTURE

(PSDS)

- Simultaneous Activity By Multiple Components
- Coordinated Via Intercomponent Interaction
- Components May Be Created Or Destroyed
- Intercomponent Interaction Patterns May Change

Figure 1

DYNAMIC PROCESS MODELLING SCHEME

(DPMS)

- Components = Processes
  - Described By DYMOL Process Templates
  - Ports
  - Buffers
- Interaction = Message Transmission
  - Finite Set Of Distinct Message Types
  - Process Ports Connected By Channels
  - Message Transmission Mediated By Links
- Dynamic Structure → Instantaneous Configurations
  - Configuration Matrix
  - Process States
  - Link States
- DPMS Model = Set Of Process Templates + Initial Instantaneous Configuration
- Computation = Sequence Of Instantaneous Configurations
  - Possibilities Determined By DYMOL Semantics

Figure 2

PARALLEL SYSTEMS WITH DYNAMIC CONNECTIVITY

(DC SYSTEMS)

- Fixed Set Of Components
- Varying Component Interaction Patterns
- Examples:
  - Computer Networks
  - Intertask Message Routing

Figure 3

DPMS MODELS OF DC SYSTEMS

- Fixed Set of Processes
- Varying Channel Connectivity
- DC DYMOL
  - No CREATE Or DESTROY
- Fixed-Size, Square, Configuration Matrices

Figure 4

CONSTRAINED EXPRESSIONS

- A Behavior Representation Technique
- Finite Expression Representing Possibly Infinite Set Of Behaviors
- Descendant Of
  - Event Expressions (Riddle)
  - Counter Expressions (Welter)
- Event Alphabet E
- Constraint Alphabet $S=S_1 \cup S_2 \cup \ldots \cup S_n$
  - $E \cap S = \emptyset$
- Event Expression Operators
  - Regular Expression Operators Plus $\square$ And $+$
- Constrained Expression Is:
  - A Regular Expression (Including $\square$) Over $E \cup S$
  - Interpreted With Respect To Constraint Set CS
- Interpreted Language L For Expression R Is:

$$L=H(R \cap C_1 \cap C_2 \cap \ldots \cap C_n)$$

where

$$CS=\{C_1, C_2, \ldots, C_n\}$$

$$H: E \cup S \rightarrow E \qquad H(e)=e \qquad H(s)=\lambda$$

Figure 5

USING HDM FOR EMBEDDED SYSTEMS*

L. Robinson and K. N. Levitt
SRI International

A generally accepted set of concepts--abstraction, hierarchical structure, and modularity--has emerged from recent software research. If followed, these concepts can lead to software with improved reliability, improved maintainability, and lower costs. These concepts have provided the basis for several informal software-development methodologies. The informal software methodologies have not produced improvements because they provide only guidelines. They neither provide a metric for evaluating a system's adherence to guidelines nor allow a mathematical proof that a system meet its requirements.

In contrast, HDM (Hierarchical Development Methodology), a formal methodology developed at SRI International, overcomes the shortcomings of the informal methodologies, because it is based on a formal model of computation and because it requires formal written specifications that describe software development decisions.

Derived from the concepts, the formal model of computation provides mechanisms with enforceable rules. These rules restrict the structure of a system and the process of its development.

HDM divides the software development process into stages--structuring, design, representation, and implementation--each of which produces formal specifications. Languages are provided for specifying the decisions at each stage; on-line tools check these specifications for violations of the rules. The consistency of these specifications can be shown through formal verification. For example, we can verify that a system's implementation meets its requirements. Thus HDM introduces tools and formal verification into all stages of system development--a first attempt to incorporate formality and checking into the design stages.

The status of HDM is twofold. It is continuing to evolve as more issues are covered and more is learned about the software development process. At the same time it has shown its value as it currently exists in developing certain classes of large systems.

SRI is developing tools to assist in the development and verification of software systems, particularly those concerned with embedded systems. SRI will apply the tools in verifying the hardware-fault-tolerance properties of SIFT (Software-Implemented Fault Tolerance), an embedded system for aircraft control in which most mechanisms for establishing tolerance to hardware faults are part of the operating system software.

*The work on the development aspects of HDM is being supported by the Naval Ocean Systems Center under Contract N00123-76-C-0195. The verification system work and its application to SIFT is being sponsored by NASA-Langley Research Center under Contract NAS 1-15528.

HDM (HIERARCHICAL DEVELOPMENT METHODOLOGY)

CONCEPTS ON WHICH HDM IS BASED

- Hierarchical Structure
- Abstraction
- Modularity
- Formal Specification
- Data Representation
- Program Verification

MECHANISMS THAT UNIFY THE CONCEPTS

- Abstract Machines: operations, internal data structures
- Modules
- Stages of development
- System families

LANGUAGES OF HDM

- HSL (Hierarchy Specification Language)--structure of machines and modules
- SPECIAL (SPECIfication and Assertion Language)--specification and data representation
- ILPL (Intermediate Level Programming Language)--operation implementation

TOOLS OF HDM

- Syntactic Checking
  - Self-consistency
  - Mutual Consistency
- Formal Verification
  - Design
  - Implementation

Figure 1

IMPACT OF HDM

ON RELIABILITY

- Production of "simpler" systems
- Feasibility of verification
- Elimination of many errors by use of languages and tools
- Feasibility of producing complete specifications
- Structuring of program testing

ON DEVELOPMENT PROCESS

- Structuring of decisions according to stages
- "Dialogues" using formal specifications
- Use of "off-the-shelf" modules
- Generality from system family approach
- Usefulness of tools in all stages

ON LIFE CYCLE MAINTENANCE

- Precise, readable documentation from specification
- Facilitation of modification

Figure 2

TOOLS OF HDM

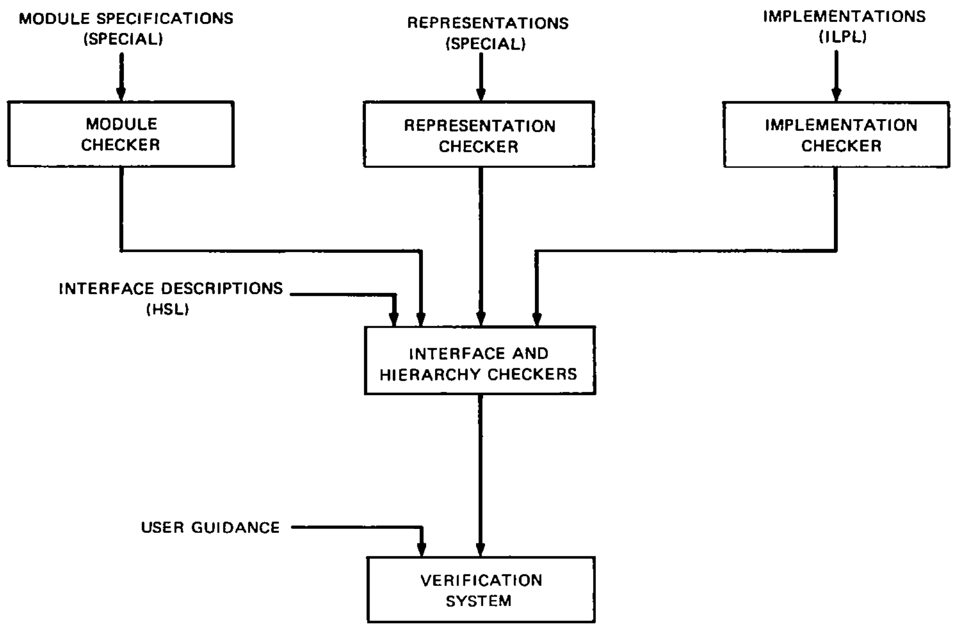

MODULE SPECIFICATIONS (SPECIAL)
REPRESENTATIONS (SPECIAL)
IMPLEMENTATIONS (ILPL)
MODULE CHECKER
REPRESENTATION CHECKER
IMPLEMENTATION CHECKER
INTERFACE DESCRIPTIONS (HSL)
INTERFACE AND HIERARCHY CHECKERS
USER GUIDANCE
VERIFICATION SYSTEM

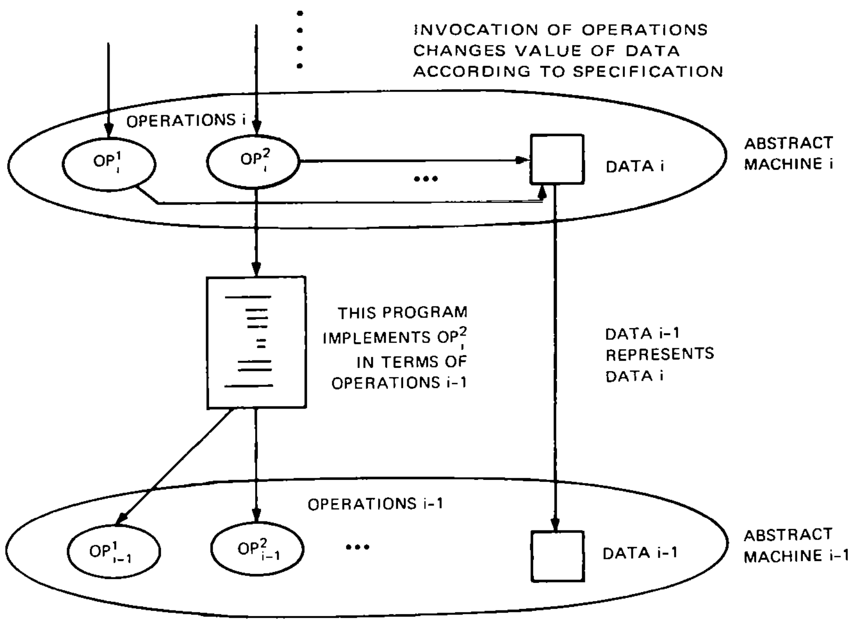
Figure 3

HIERARCHY FOR PSOS (Provably Secure Operating System)
(Only a subset is shown)

L9	.	USER PROCESSES	.	VISIBLE I/O	.	.	.	.
L8	.	.	.	.	.	.	.	USER OBJECTS
L7	.	.	.	.	.	.	DIRECTORIES	
L6	.	.	.	.	.	EXTENDED TYPES		
L5	.	.	SEGMENTS	.	.			
L4	.	.	PAGES	.	.			
L3	.	SYSTEM PROCESSES	.	SYSTEM I/O	.			
L2	.	.	.	.	ARITHMETIC			
L1	.	.	MEMORY	PRIMITIVE I/O				
L0	CAPABILITIES	INTERRUPTS						

Figure 4

ADJACENT ABSTRACT MACHINES IN A HIERARCHY



INVOCATION OF OPERATIONS CHANGES VALUE OF DATA ACCORDING TO SPECIFICATION
OPERATIONS i
OP^1_i
OP^2_i
DATA i
ABSTRACT MACHINE i
THIS PROGRAM IMPLEMENTS OP^2_i IN TERMS OF OPERATIONS i-1
DATA i-1 REPRESENTS DATA i
OPERATIONS i-1
OP^1_{i-1}
OP^2_{i-1}
DATA i-1
ABSTRACT MACHINE i-1

Figure 5

STAGES OF HDM

STAGE	ACTIVITY	LANGUAGE	TOOL(S)
Conceptualization	Formulating system goals	None yet	None yet
External Interface Definition	Defining external and module structure of extreme machines	HSL	Interface checker
Intermediate Interface Definition	Defining external and module structure of intermediate machines	HSL	Interface and Hierarchy checkers
Formal Specification	Specifying modules	SPECIAL	Module checker
Formal Representation	Describing data structures	SPECIAL	Representation checker
Abstract Implementation	Writing implementation specifications	ILPL	Implementation checker
Coding	Producing executable programs	A programming language	Compilers, Optimizers, Assemblers, etc.
Verification	Proving properties of formal descriptions	Language of verification system	Verification system

Figure 6

AN INTEGRATED VERIFICATION AND VALIDATION TOOL FOR FLIGHT SOFTWARE

Richard N. Taylor, Leon J. Osterweil*, and Leon G. Stucki

Boeing Computer Services Company

NASA Langley Research Center is developing the MUST (Multipurpose User-oriented Software Technology) program to cut the cost of producing research flight software through a system of software support tools. Boeing Computer Services Company (BCS) has designed an integrated verification and validation capability as part of MUST. Documentation, verification and testing options are provided with special attention on real-time, multiprocessing issues. The needs of the entire software production cycle have been considered, with effective management and reduced lifecycle costs as foremost goals.

Previous verification and validation systems generally have utilized a single technique, such as static or dynamic analysis. However, thorough examination of any one program requires the use of several techniques. Besides providing a comprehensive set of analytical techniques, the integrated capability BCS has designed takes advantage of the complementary abilities of the different schemes in a synergistic manner. A "one-tool-does-it-all" concept has not emerged though. The need for a distributed set of tools became clear as the various usage modes present in the MUST environment were modeled. No single sequence of testing and analysis activities is optimally suited to all MUST requirements. Rather, for detecting specific classes of errors under specific operating constraints, a specific combination of analysis techniques is chosen.

The concern with multiprocessing issues is motivated by the increasing sophistication of flight hardware and software, which present difficulties such as protecting shared data. New research was conducted into the problem of statically detecting such errors with encouraging results. Consequently, capabilities have been included in the design for static detection of data flow anomalies involving communicating concurrent processes. Some types of ill-formed process synchronization and deadlock also are detected statically.

Although the HAL/S language is the primary subject of this design, the algorithms developed are readily applicable to other languages. Prototype capabilities for HAL/S have been developed in conjunction with the design. Full implementation of these capabilities will provide the MUST user with extremely powerful program development tools. Such programming environments offer a very desirable and profitable alternative to the way software is typically produced.

*Boeing Computer Services Company and the University of Colorado

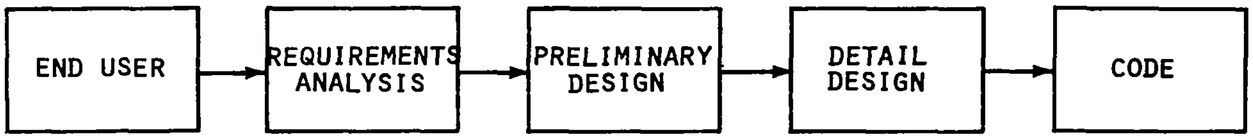


Figure 1

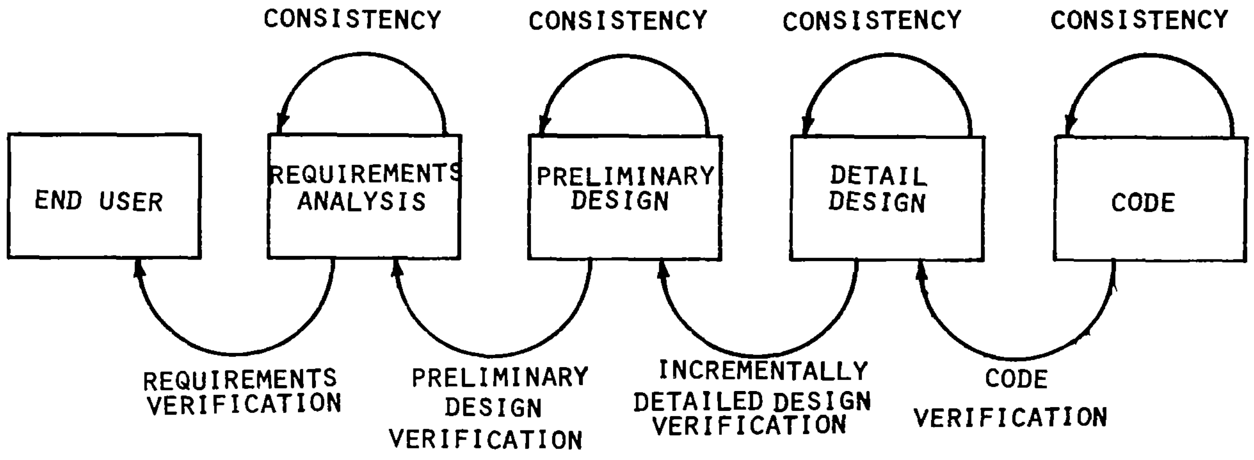


Figure 2

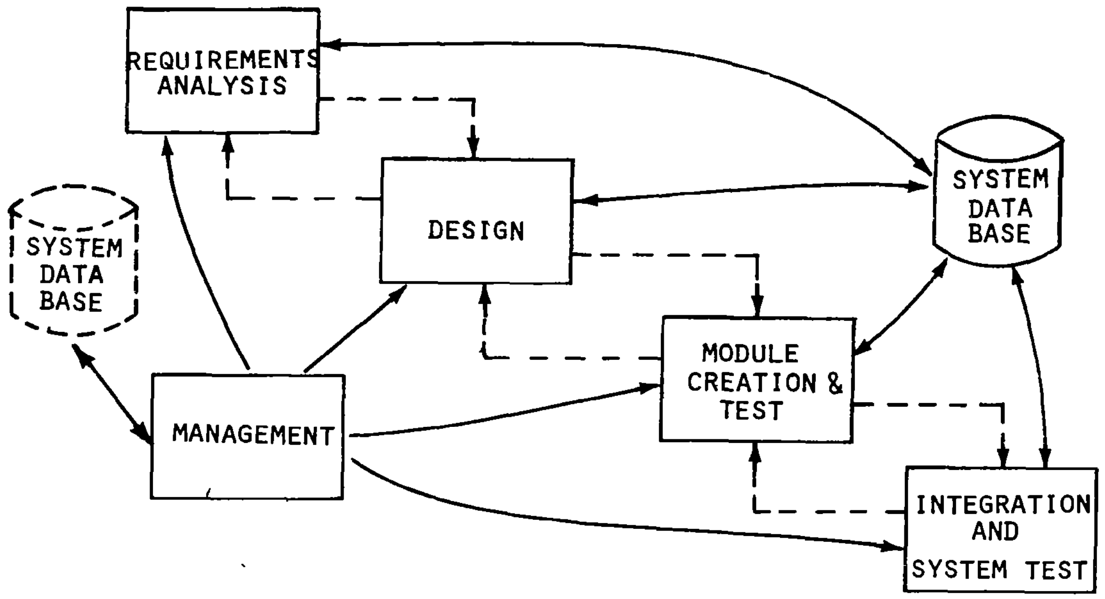


Figure 3

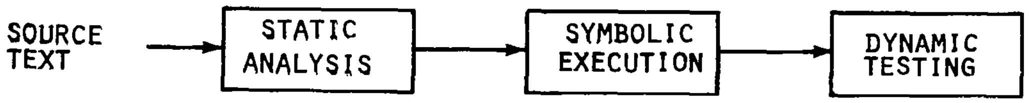
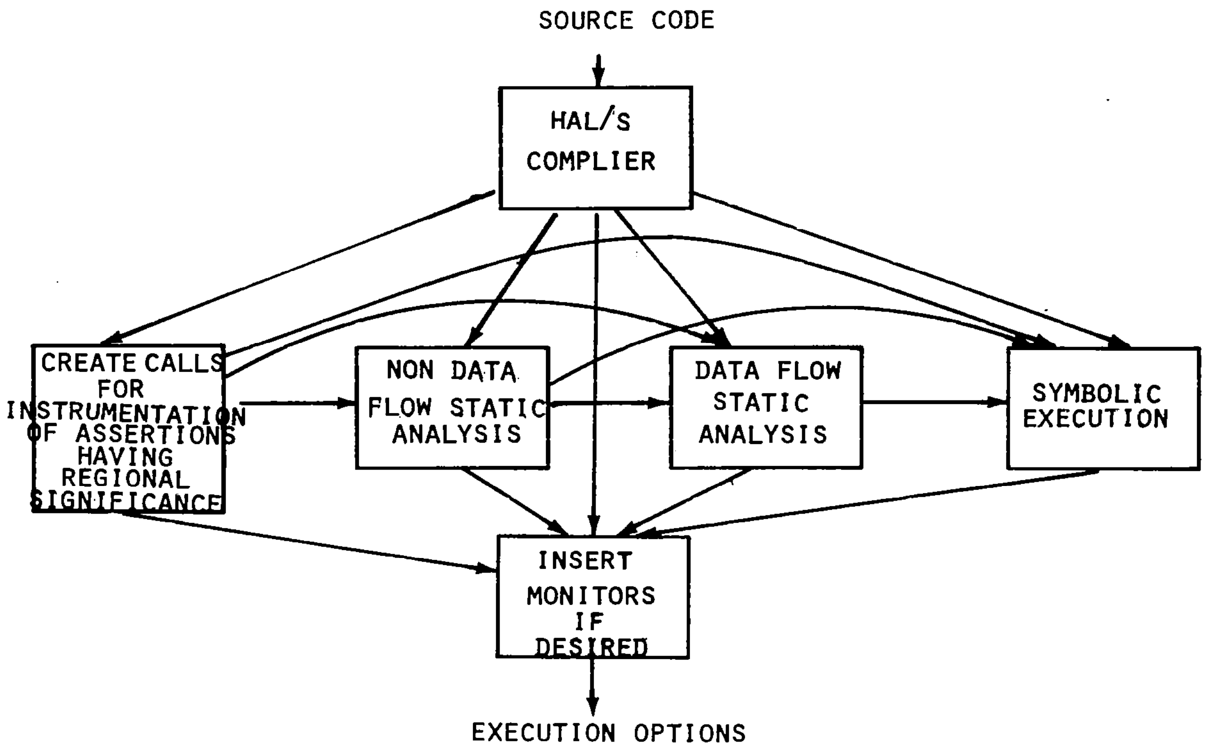


Figure 4


SOURCE CODE
HAL/S COMPLIER
CREATE CALLS FOR INSTRUMENTATION OF ASSERTIONS HAVING REGIONAL SIGNIFICANCE
NON DATA FLOW STATIC ANALYSIS
DATA FLOW STATIC ANALYSIS
SYMBOLIC EXECUTION
INSERT MONITORS IF DESIRED
EXECUTION OPTIONS

MODULE VERIFICATION OPTIONS

Figure 5

CONCLUSIONS

1) PROGRAMMING ENVIRONMENTS A SIGNIFICANT AID IN THE PRODUCTION OF SOFTWARE

2) A COMPREHENSIVE SET OF INTEGRATED VERIFICATION AND VALIDATION TOOLS ALLOW A MAXIMUM AMOUNT OF TESTING TO BE PERFORMED IN THE PROGRAMMING ENVIRONMENT

3) SOFTWARE LIFECYCLE ISSUES MAY BE EFFECTIVELY ADDRESSED

RECOMMENDATIONS

1) IMPLEMENTATION AND UTILIZATION OF PROGRAMMING ENVIRONMENTS

2) MORE RESEARCH INTO THE DEVELOPMENT OF INTEGRATED REQUIREMENTS AND DESIGN TOOLS. ALLOW VERIFICATION BETWEEN VARIOUS LEVELS OF SPECIFICATIONS

3) FURTHER RESEARCH INTO VERIFICATION OF REAL TIME, CONCURRENT PROCESS SOFTWARE

Figure 6

USE OF SYMBOLIC EXECUTION IN VERIFICATION AND VALIDATION

Marilyn S. Fujii and Michael A. Ikezawa
Logicon, Inc.

INTRODUCTION

AMPIC is a symbolic execution tool used in verification and validation of assembly and higher order language programs. AMPIC has three major processing phases: structure analysis, expression translation, and path analysis. Each of these phases is invoked sequentially to perform a portion of the structural calculus required for symbolic execution.

STRUCTURE ANALYSIS

The first phase, structure analysis, decomposes the program into two basic types of structural elements referred to as sequences and transfers. A sequence is an ordered group of executable statements or instructions that must be followed consecutively from top to bottom. A transfer is any program statement or instruction that causes the selection of the next statement or instruction from two alternatives. In the sample program shown in figure 1, the sequence and transfer segments are indicated by consecutively numbered S and T symbols.

The structure analysis phase next produces an equivalent, well-structured representation of the program which is displayed in either of two types of flowcharts. The abbreviated flowchart, shown in figure 2, concisely summarizes the program's structure in terms of its S and T segments. The full text flowcharts are similar, but replace S and T symbols with actual source code.

The structured flowcharts are used in verification and validation to reveal distinct program paths, a significant advantage over conventional flowcharts. Comparing structured flowcharts to design specifications detects errors in implemented program logic. Additionally, segments occurring in more than one path can be identified as candidates for optimization.

EXPRESSION TRANSLATION

The second phase, expression translation, transforms the program's source code into a mathematical notation. Particularly for assembly language programs, expression translation minimizes the amount of painstaking and error-prone manual analysis. The translations for higher order language programs resemble the original code, with some algebraic simplification. As shown in the translation of the sample program (fig. 3), the symbol NU indicates the final value of a variable within each segment.

Expression translation is most useful in the verification and validation of assembly language programs, for which it is necessary to translate source code into a less machine-oriented form. Figure 4 shows the assembly language equivalent of sequence S2 from the sample program and its translation. In comparison to the source code, the translation is quickly comprehended and can easily be checked against equations in the design specification.

PATH ANALYSIS

The third phase, path analysis, identifies each program path as a series of S and T segments. The conditions that are logically necessary to execute each path are derived, as are the results of executing each path. Figure 5 shows the symbolic execution results for two of the sample program's paths. For convenience in describing paths, the transfer outcome is indicated by replacing the T transfer symbol with a P (no transfer) or an R (transfer).

AMPIC's path analysis is used in verification and validation to derive the initial and resulting conditions for program paths. It reveals unreachable code by identifying contradictory initial conditions. For programs suitable for case-by-case analysis, path conditions and results are often readily comparable to design specifications. Path analysis results can also be used to systematically generate test cases that provide complete coverage.

SUMMARY

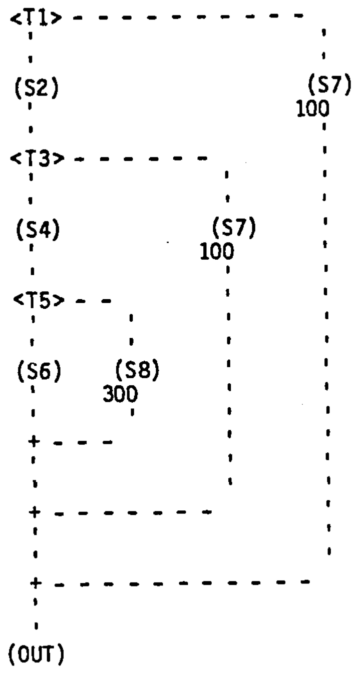
AMPIC has been successfully applied on several verification and validation projects including tracking, avionics flight control, electronic countermeasures, command and control, and targeting applications. Our experience in applying AMPIC has shown that its capability to analyze complex programs has reduced the need for several commonly employed software tools and has added much needed rigor to verification and validations techniques.

```
      SUBROUTINE ROOT
      IMPLICIT INTEGER*2(I,K-N)
      COMMON/ARGS/A,B,C
      COMMON/VALUE/ROOT1,ROOT2,NROOT
      IF(A.EQ.0) GOTO 100
      TEMP1 = 4.*A*C
      TEMP1 = B*B - TEMP1
      IF(TEMP1.LT.0) GOTO 100
      TEMP2 = 2.*A
      ROOT2 = -(B/TEMP2)
      IF(TEMP1.EQ.0) GOTO 300
      TEMP2 = FSQRT(TEMP1)/TEMP2
      ROOT1 = ROOT2 + TEMP2
      ROOT2 = ROOT2 - TEMP2
      NROOT = 2
      RETURN
100   NROOT = 0
      RETURN
300   ROOT1 = ROOT2
      NROOT = 1
      RETURN
      END
```

```
T1   IF(A.EQ.0.) GOTO S7

S2   TEMP1=4.*A*C
     TEMP1=B*B-TEMP1

T3   IF(TEMP1.LT.0.) GOTO S7

S4   TEMP2=2.*A
     ROOT2=-B/TEMP2

T5   IF(TEMP1.EQ.0.) GOTO S8

S6   TEMP2=FSQRT(TEMP1)/TEMP2
     ROOT1=ROOT2+TEMP2
     ROOT2=ROOT2-TEMP2
     NROOT=2
     GOTO OUT

S7   NROOT=0
     GOTO OUT

S8   ROOT1=ROOT2
     NROOT=1
     GOTO OUT
```

Sample Program and its Sequence and Transfer Segments

Figure 1



Abbreviated Flowchart of Sample Program

Figure 2

```
T1:  JUMP      (A = 0.)
     NO JUMP   (A ≠ 0.)

S2:  TEMP1<NU> = B * B - 4. * A * C

T3:  JUMP      (TEMP1 < 0.)
     NO JUMP   (TEMP1 ≥ 0.)

S4:  TEMP2<NU> = 2. * A
     ROOT2<NU> = - B / (2. * A)

T5:  JUMP      (TEMP1 = 0.)
     NO JUMP   (TEMP1 ≠ 0.)

S6:  TEMP2<NU> = .FSQRT(TEMP1) / TEMP2
     ROOT1<NU> = ROOT2 + .FSQRT(TEMP1) / TEMP2
     ROOT2<NU> = ROOT2 - .FSQRT(TEMP1) / TEMP2
     NROOT<NU> = 2

S7:  NROOT<NU> = 0

S8:  ROOT1<NU> = ROOT2
     NROOT<NU> = 1
```

Translation of Sample Program

Figure 3

```
S2  DLA   OC1            S2:  $A<NU> = B * B - 4. * A * C
    FMR   A                   $Q<NU> = 0
    FMR   C                   $CD<NU> ≠ 1
    DSA   TEMP1               TEMP1<NU> = B * B - 4. * A * C
    DLA   B
    FMR   B
    FANR  TEMP1
    DSA   TEMP1
```

Translation for Equivalent Assembly Language Segment

Figure 4

```
PATH:  (P1,S2,P3,S4,P5,S6,OUT)

IF:    (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C ≠ 0.)

THEN:  TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT2<NU> = - B / (2. * A) - .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT1<NU> = - B / (2. * A) + .FSQRT(B * B - 4. * A * C) / (2. * A)
       NROOT<NU> = 2
```

```
PATH:  (P1,S2,P3,S4,R5,S8,OUT)

IF:    (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C = 0.)

THEN:  TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = 2. * A
       ROOT2<NU> = - B / (2. * A)
       ROOT1<NU> = - B / (2. * A)
       NROOT<NU> = 1
```

Symbolic Execution Results for Two Paths of Sample Program

Figure 5

SQLAB - Tools for Program Verification

Sabina H. Saib
General Research Corporation

The Software Quality Laboratory (SQLAB) is made up of a collection of tools which can assist in the verification of a program written in one of several programming languages (FORTRAN, IFTRAN, PASCAL, and Verifiable PASCAL). The tools available in SQLAB provide reports in a fashion similar to compiler diagnostics on errors that have been found to be costly in a number of error studies.

Two major types of analyses are available: those that require additional information prepared when the program is written and those that can analyze the program as it is written normally. The set/use, mode, infinite loop, external reference, and unreachable code analyses require no additional statements. The asserted use, units, and consistency proofs require additional information in the form of assertions.

The following capabilities are available for statically determining consistency between source-level specifications and source code:

1. Verification condition generation by symbolically executing INITIAL, ASSERT, and FINAL statements in combination with source-code statements

2. Logical simplification of verification conditions by applying standard normalization and simplification rules of predicate calculus and first-order logic, as well as user-supplied axioms

3. Data access correctness checking of asserted access rights to non-local data and parameters and actual access based on data flow analysis

4. Units correctness checking by automatically comparing embedded physical unit specifications with computational, decision, and procedure reference statements

In addition, multi-module documentation reports, parameter checking reports, and automatic instrumentation are available.

SOFTWARE QUALITY LABORATORY

- INTRODUCE PRACTICAL APPLICATIONS OF PROOF OF CORRECTNESS TO LARGE, COMPLEX, REAL TIME PROGRAMS
- ELIMINATE COMMON ERRORS FROM SOFTWARE
- INVESTIGATE LANGUAGES AND LANGUAGE CONSTRUCTS WHICH AID SOFTWARE QUALITY

Figure 1

SQLAB CAPABILITIES

STATIC ANALYSIS - WITHOUT ASSERTIONS

STATIC ANALYSIS - WITH ASSERTIONS

EXECUTION OF ASSERTIONS

VERIFICATION CONDITION GENERATION

COVERAGE ANALYSIS

Figure 2

FUTURE PLANS FOR SQLAB

- DEMONSTRATE DETECTION OF SPECIFIC ERROR TYPES
  IN LARGE PROGRAMS

- PROVIDE NEW ASSERTION TYPES
  SEQUENCE OF OPERATIONS
  PRIORITY AND TIMING

- PROPOSE SPECIFICATION LANGUAGE
  TIE THE MODULE STUBS TO SPECIFICATION

Figure 3

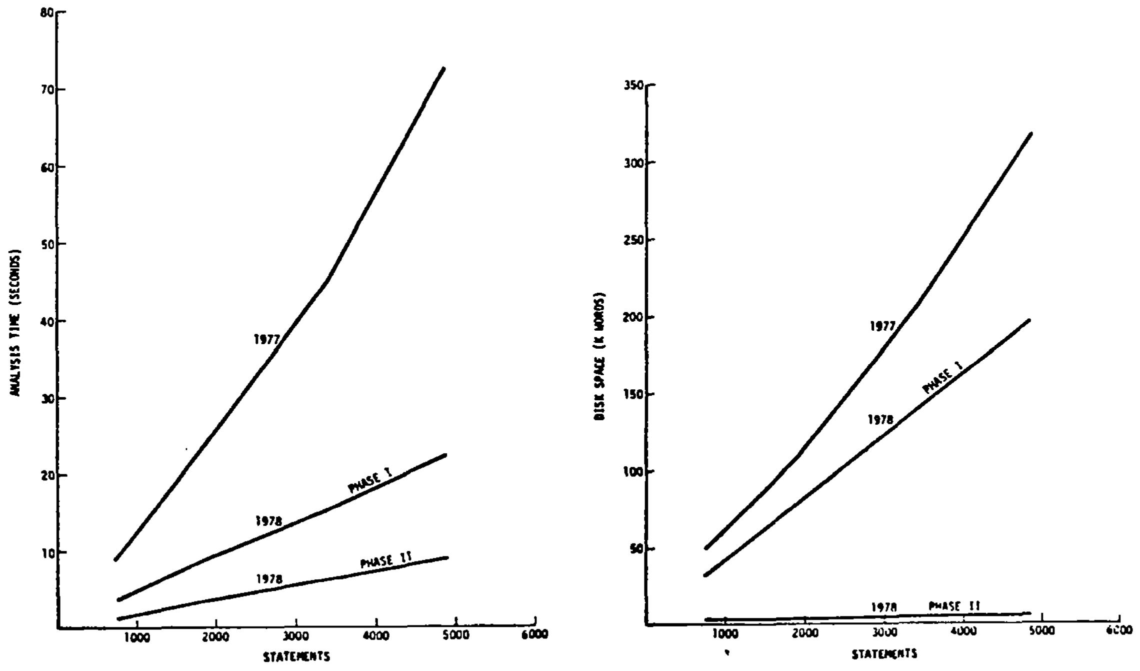
```
LOOP ANALYSIS                              SUBROUTINE TEST
THE CONDITION THAT MUST BE TRUE FOR THIS LOOP TO TERMINATE IS-
              KFLAG .GT. 0 .AND. K .GT. 0

   4                  WHILE ( I .LE. N )
   5 ( 1)             .  IF ( I .EQ. 5 )
   6 ( 2)             .  .  I = I + K
   7 ( 1)             .  ELSE
   8 ( 2)             .  .  I = I + KFLAG
   9 ( 1)             .  ENDIF
  10                  ENDWHILE
```

Figure 4

AQA CONTRIBUTIONS VS STATE OF THE ART

● PRACTICAL USE OF ASSERTION TECHNIQUES	VS	SOLE DEPENDENCE ON PROOF
● REAL PROGRAMMING LANUAGES FORTRAN PASCAL	VS	THEORETICAL LANGUAGES LISP NUCLEUS
● NORMAL ARITHMETIC REAL DOUBLE PRECISION COMPLEX	VS	ONLY INTEGER
● NORMAL DATA STRUCTURES ARRAY	VS	ONLY SIMPLE VARIABLES
● RECOVERY FROM FAULTS	VS	CATASTROPHIC FAILURE
● ANALYSIS OF LARGE PROGRAMS WITH REASONABLE USE OF COMPUTER RESOURCES	VS	IMPRACTICAL COMPUTATION TIMES

Figure 5



NO. STATEMENTS VS TIME

NO. STATEMENTS VS DISK MEMORY

Figure 6

A SOFTWARE QUALITY ASSURANCE EXPERIMENT*

J. P. Benson
S. H. Saib

General Research Corporation

Over the past two years a number of techniques designed to eliminate errors in software have been implemented in a collection of programs called the Software Quality Laboratory. An important goal of this effort has been the ability to analyze "realistic" programs. By *realistic* we mean programs which can execute on current computers with current compilers, use floating point arithmetic, incorporate data structures, be composed of multiple modules, and have a total size of perhaps several thousand statements.

In order to demonstrate SQLAB's ability to locate errors, a medium size program (~1000 statements) was selected. The program simulates the tracking of objects by a radar and embodies many of the characteristics of a complex software system including multitasking and data structures composed of queues and records.

The experiment described in this paper was designed to evaluate the use of assertions in a real time program. The experiment consisted of adding errors to the test program from a list of the most common software errors. A number of errors from a set of error categories were selected and introduced into the test program. (During the course of the experiment some errors already present in the program were also detected.) Executable assertions were written to detect the errors and the program was run to verify that the errors were actually detected. The results suggest that some of the assertions could have been made part of the variable definition statements of the programming language itself rather than separate statements. In addition, three new types of assertions which would be useful in error detection were identified: variable range assertions, approximate result assertions, and sequencing assertions.

---

* This work was supported by the Army Ballistic Missile Defense Systems Command under contract DASG60-76-C-0050.

SOFTWARE VERIFICATION USING SQLAB

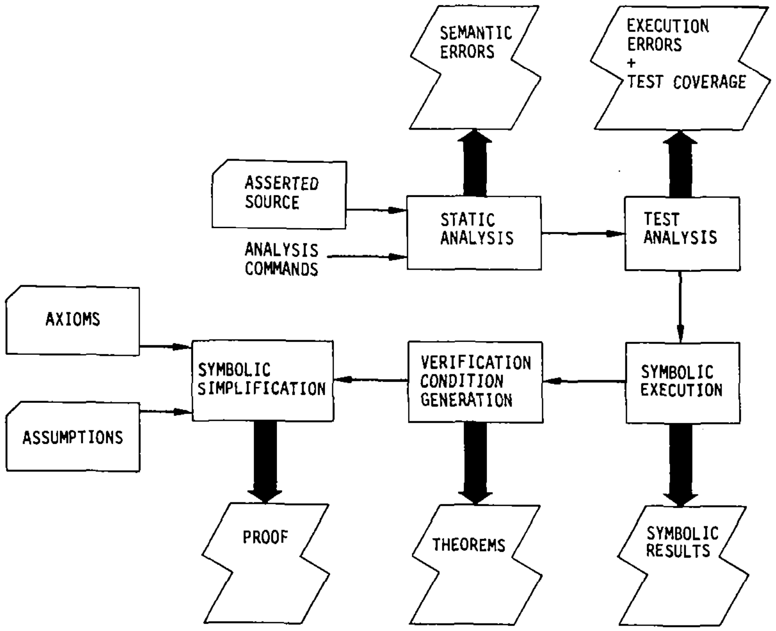


Figure 1

USES OF ASSERTIONS

- PROOF OF CORRECTNESS

```
INITIAL  (B >= 0);
D := B;
C := 0;
WHILE (D ≠ 0);
ASSERT (C + A * D = A * B);
∴ (B >= 0) AND (D ≠ 0) => (C + A * B = A * B)
```

- EXECUTION TIME VALIDATION

```
ASSERT ((RFIRST > IR) AND (XNEXT > 1X) AND (XLATE <= IR));
FAIL
     TIMEOVERLAP
END FAIL;
```

- STATIC ANALYSIS

```
VAR SOVEL : REAL UNITS METERS / USEC;
INPUT RETURN, SPOUT, OBJID;
OUTPUT OTDSREC, TOTDSREC, SRDSREC;
```

Figure 2

SOFTWARE ERROR CATEGORIES

COMPUTATION ERRORS
- USING THE WRONG EQUATION
- OVERFLOW
- UNDERFLOW
- MISSING COMPUTATION
- EXTRANEOUS COMPUTATION

DATA HANDLING ERRORS
- SUBSCRIPT ERRORS
- FAILURE TO INITIALIZE A VARIABLE
- REFERENCING THE WRONG VARIABLE
- UPDATING THE WRONG VARIABLE

LOGIC ERRORS
- MISSING TEST
- INCORRECT TEST
- INCORRECT SEQUENCING

Figure 3

SOFTWARE QUALITY EXPERIMENT

<u>HYPOTHESIS</u>: EXECUTABLE ASSERTIONS ARE MORE EFFICIENT AND EFFECTIVE THAN CORRECTNESS PROOF OR STATIC ANALYSIS IN DETECTING THE MOST COMMON TYPES OF PROGRAM ERRORS.

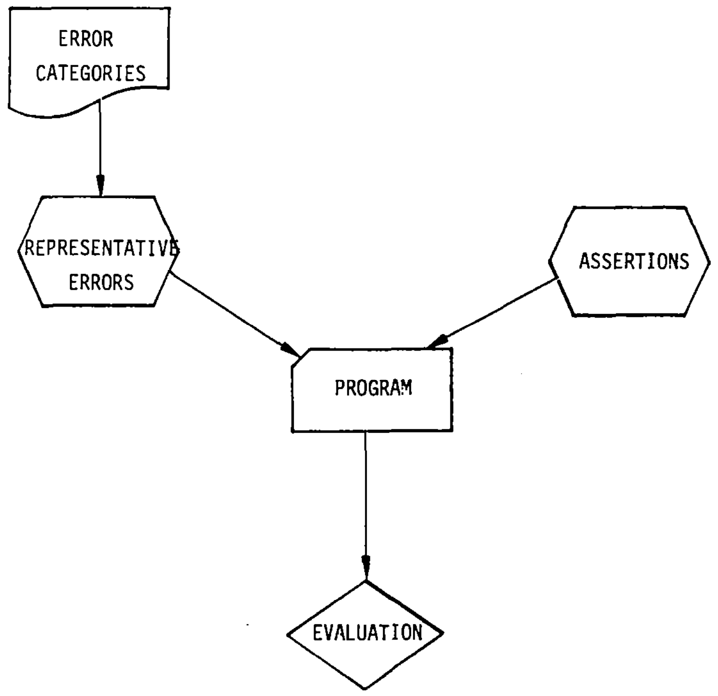


Figure 4

REPRESENTATIVE ERRORS

COMPUTATION ERRORS

C1:	USING THE WRONG VARIABLE NAME IN AN EQUATION
C2:	LEAVING OUT A COMPUTATION
C3:	ADDING AN UNNEEDED COMPUTATION

DATA HANDLING ERRORS

D1:	REFERENCING THE WRONG VARIABLE NAME
D2:	USING THE WRONG ARITHMETIC OPERATOR
D3:	NOT INITIALIZING A VARIABLE CORRECTLY

LOGIC ERRORS

L1:	LEAVING OUT A TEST
L2:	USING THE WRONG RELATIONAL OPERATOR IN A TEST
L3:	EXECUTING THE WRONG SEQUENCE OF DECISIONS

Figure 5

RESULTS AND CONCLUSIONS

ERROR DETECTION METHODS

ERROR TYPE	ERROR CHECK	METHOD
C1	CASE ANALYSIS	STATIC
D1	CASE ANALYSIS	STATIC
D2	RANGES AND BOUNDS	STATIC
D3	INITIALIZATION AND RANGE CHECKS	STATIC
C2	BOUNDS	EXECUTABLE
C3	DUPLICATION	EXECUTABLE
L1	REQUIREMENTS	PROOF
L2	AUXILIARY VARIABLE (INVARIANT)	PROOF
L3	ASSERTED ELSE	PROOF

COST

PROGRAM WITH ASSERTIONS

Compilation Time	+56%
Execution Time	+12%
Load Length	+13.5%

Figure 6

A SIMULATOR DEVICE FOR VALIDATION OF GENERAL AVIONICS EMBEDDED SOFTWARE

Byron M. Allen and Gary H. Barber
Intermetrics, Inc.

SUMMARY

The recent trends in avionics processing have been towards distributed digital computers. This trend has led to a proliferation of computers connected in a complex fashion. This situation has resulted in serious integration problems. The history of software development has shown that integration with systems external to the given computer is the most difficult and costly portion of software development. Effective tools are needed to validate avionics software before expensive flight testing and without safety of flight restrictions.

The military avionic systems have contained digital computers since 1968 and the science of verification and validation has evolved since then. Many tools ranging from instruction level simulators to Integrated Avionic simulation have resulted and today a full range of these tools are utilized to verify the operational flight programs of today's military aircraft.

The software required to drive these simulators has also evolved since running in real time is often required. Aircraft models come in varying complexities depending upon the avionic device being tested.

Since general aviation avionic manufacturers do not normally have the extended budgets required for exhaustive levels of testing, a less expensive generalized tool is required. This tool could be utilized for the development and testing of flight software as well as extensive hardware/software integration. This tool should provide all basic requirements of system testing prior to flight test. The tool is designed in a modular fashion such that both hardware and software can be modified to provide testing of a wide variety of avionic systems.

DEFINITION OF DISTRIBUTED PROCESSING

- MULTIPLICITY OF RESOURCES
- PHYSICAL DISTRIBUTION
- COOPERATIVE AUTONOMY
- CONTROL ARCHITECTURE
- SYSTEM EXECUTIVE
- SYNCHRONIZATION SCHEME

Figure 1

HISTORY OF MILITARY AVIONIC SOFTWARE

- o A7A, A7B ANALOG SYSTEMS WERE REPLACED BY A CENTRALIZED DIGITAL COMPUTER IN 1969
- o AVIONIC SYSTEM QUICKLY OUTGREW COMPUTER CAPABILITIES AND SOFTWARE SUFFERED
- o HIGHER ORDER LANGUAGES FOR AVIONICS WERE REQUIRED IN ORDER TO INCREASE UNDERSTANDABILITY
- o INVESTMENTS IN SIMULATORS OF MANY LEVELS HAVE BEEN DEVELOPED FOR EVERY AIRCRAFT SYSTEM

Figure 2

GENERAL AVIATION PROBLEMS

- CERTIFICATION OF EMBEDDED SOFTWARE WILL BE A CONTINUING PROBLEM
- GENERAL AVIATION AVIONIC MANUFACTURERS DO NOT HAVE EXTENSIVE BUDGETS THAT ALLOW PURCHASE OF MULTI-LEVELS OF SIMULATION EQUIPMENT
- PURCHASE OF MULTIPLE COMPUTER SYSTEMS THAT SPECIALIZED FOR PARTICULAR TASKS IS NOT FEASIBLE
- CONCLUSION - AN INTEGRATED AVIONIC SIMULATOR THAT SUPPORTS SOFTWARE DEVELOPMENT, SOFTWARE TEST, AND HARDWARE INTEGRATION IS REQUIRED

Figure 3

BLOCK DIAGRAM OF INTEGRATED AVIONIC SIMULATION

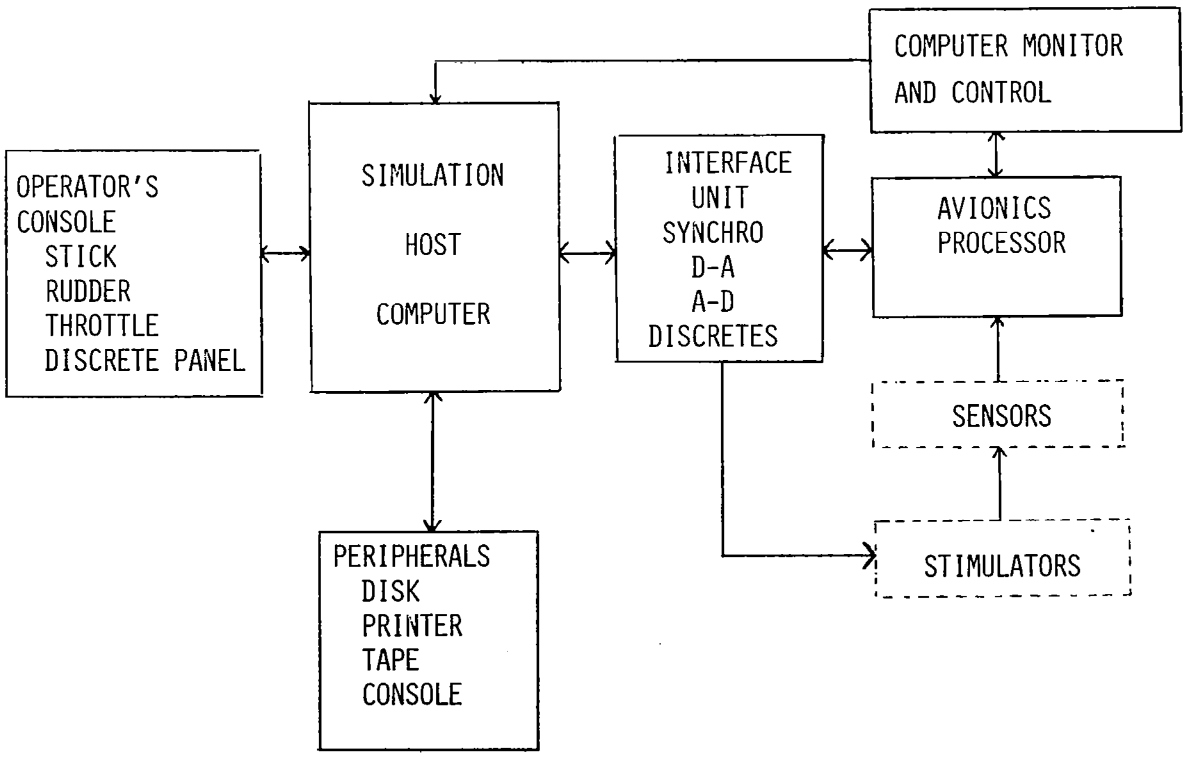


Figure 4

IAS SOFTWARE

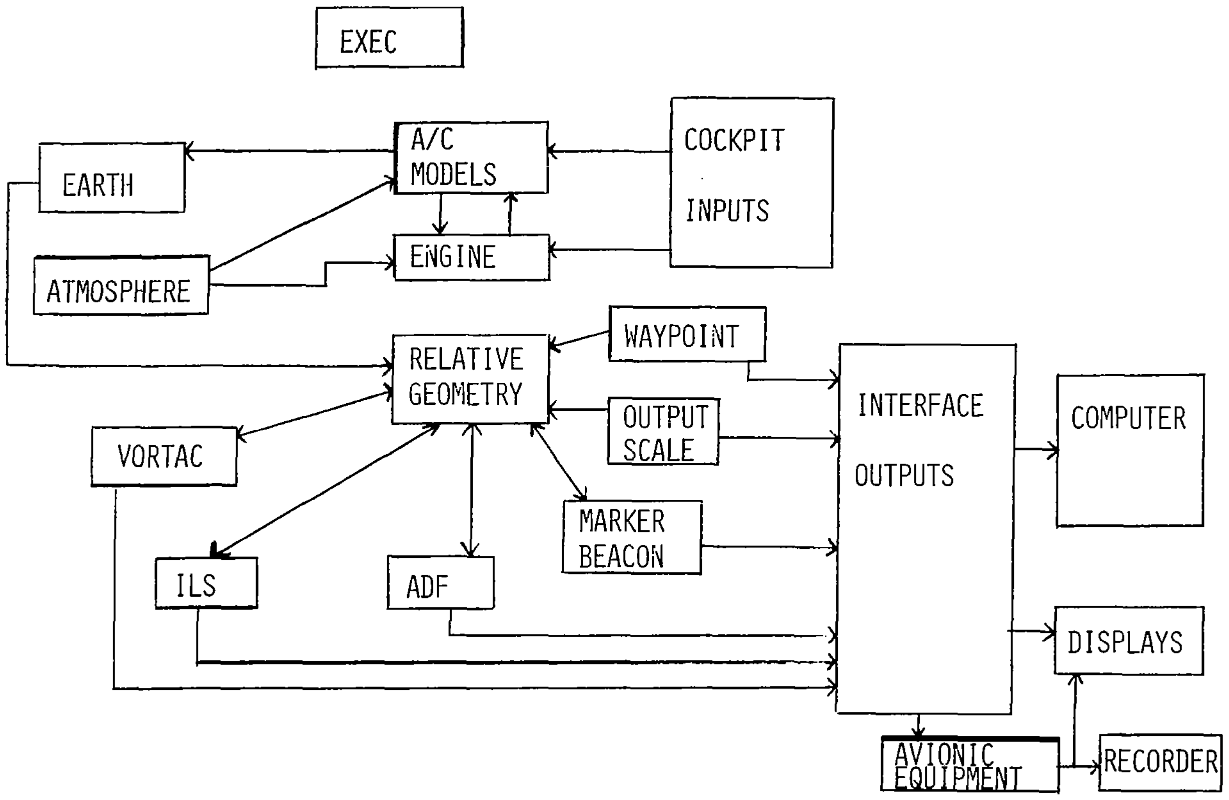


Figure 5

INTEGRATED AVIONIC SIMULATOR

- UTILIZES MUCH OF THE SAME EQUIPMENT AS THE DTS EXCEPT IT PROVIDES INTERFACES FOR USE OF ACTUAL AVIONIC EQUIPMENT
- HIGH SPEED CONTROLLER PROVIDES PROGRAM SUPPORT REQUIRED FOR DEBUG
- THE SPECIAL INTERFACE DEVICE PROVIDES A GENERAL INTERFACING CAPABILITY FOR VARIOUS ANALOG AND DISCRETE SIGNALS

Figure 6

DYNAMIC INTEGRATED TEST FOR THE SPACE SHUTTLE

Saul F. Stanten
Intermetrics, Inc.

A requirement exists for integrated testing of the Space Shuttle vehicle, at Kennedy Space Center (KSC), prior to orbital flight. Vehicle complexity and cost considerations have forced a search for innovative techniques for implementation of integrated vehicle testing. This paper describes the motivation for Dynamic Integrated Test (DIT), the technique developed, and experimental evidence which verifies the soundness of the approach. In addition, the applicability to other real time avionics systems is explored. The test concept has been accepted by the NASA, and tests on the OFT-1 vehicle are planned for the summer of 1979.

KSC is the first and only place that the actual orbiter, mated elements (solid Rocket Boosters and External Tank), software, Ground Support Equipment and payloads all come together. It is essential to check the total ascent and entry configuration in an integrated fashion prior to flight. Examples of items to be verified during such a test include data bus activity patterns, critical timing sequences, software and hardware moding as a function of flight parameters, interaction between real sensors, flight software, real effectors, compatibility of ground launch software/hardware and on-board software/hardware, the absence of EMI problems, and other systems interactions which cannot be tested fully in a laboratory environment.

The Space Shuttle is a complex digitally controlled vehicle in which most navigation, guidance, flight control, systems management, sequencing, display generation, and crew controls are processed by the redundant central digital computers. To perform a truly integrated test of the mated vehicle, it is necessary to supply coordinated sensor and crew data to the flight software; so the system can mode and sequence as it does during a flight. In addition, the effect of real sensors (IMUs, TACAN, pressure transducers, crew, etc) upon real effectors (elevons, rudder, engine bells, landing gears, ventdoors, displays, etc) should be measured. Vehicle safety requirements also necessitate that certain signals such as those involved in the mixing of hypergolic fuels and oxidizers be inhibited during a DIT.

The DIT methodology allows the above requirements to be satisfied in a conceptually simple manner. A defined flight scenario is first run in a closed loop digital simulation. The sensor inputs to the simulated flight software are recorded and converted by an off line program (SIMGEN) into the proper format to drive the flight software during a DIT run. This data (SIMDATA) is then supplied to the vehicle by means of the Launch Data Bus, which is controlled by the Launch Processing System. SIMDATA may be used instead of real sensor data (substitution mode), or it may be added to real sensor data (combination mode) to produce realistic sensor profiles. The combination mode necessitates that SIMDATA be compensated with ground nominal models of the sensors.

DIT runs are evaluated by observing telemetry data and cockpit displays. The DIT data is compared against analogous results of the original digital simulation and laboratory DIT runs.

Initial development of DITs are performed in the Rockwell Avionics Development Laboratory. Shuttle Avionics Test Sets (SATS) are employed to control the tests, record telemetry data, and simulate real sensors when necessary. Real flight computers, mass memory and displays are employed. At KSC, DIT tests are planned to be run in the Orbiter Processing Facility (OPF) and in the Vehicle Assembly Building (VAB).

Successful DIT runs have been performed in the Avionics Development Laboratory on both the Approach and Landing Test software and on preliminary Orbital Flight Test software. These tests verified the described techniques: SIMDATA was combined with a real IMU sensor data to demonstrate the end to end test capability; synchronization of SIMDATA and flight software was achieved. The test have proved to be repeatable, and good agreement between DIT runs and the digital simulation runs have been achieved. Three IMU alignment discrepancies and one flight software discrepancy were discovered as a by-product of these activities.

As a result of our activities to date a number of generalizations relevant to other avionics systems may be suggested. First, consideration of an integrated test capability at the beginning of a program can prevent the unnecessary expense of retrofitting the test after the program in underway. Second, a port into the flight software must be provided so that sensor data may be injected into flight software. Third, flight software code should be provided with a test mode whereby it can accept and utilize externally supplied sensor data. Fourth, flight software should be provided with a mechanism to inhibit hazardous ouputs during an integrated test. Finally the DIT technique can be used to run any desired scenario with any degree of off nominal performance. It can be employed not only as a final integration test (as planned for KSC) but also as a more comprehensive system verification tool.

BASELINE HARDWARE CONFIGURATION

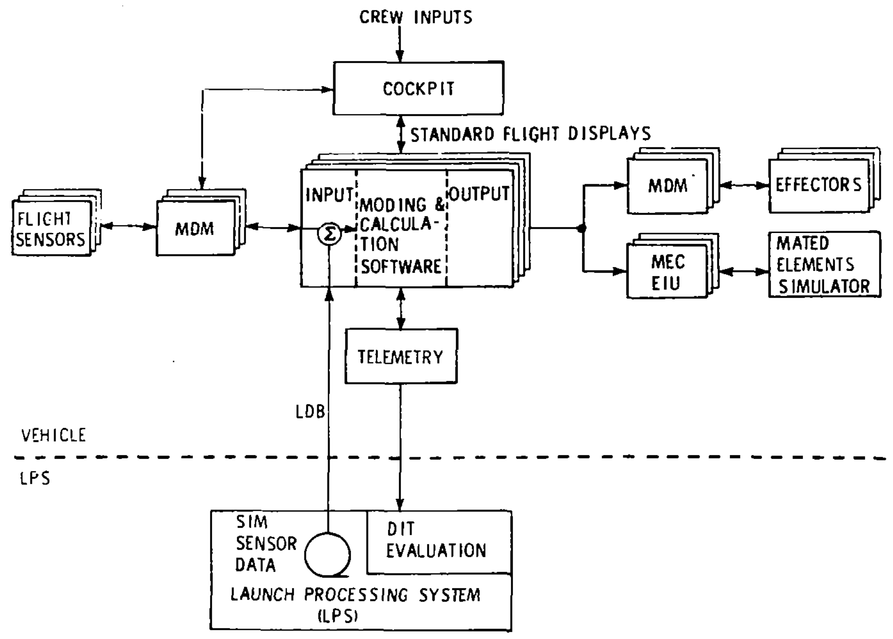


Figure 1

DIT DEVELOPMENT PROCESS

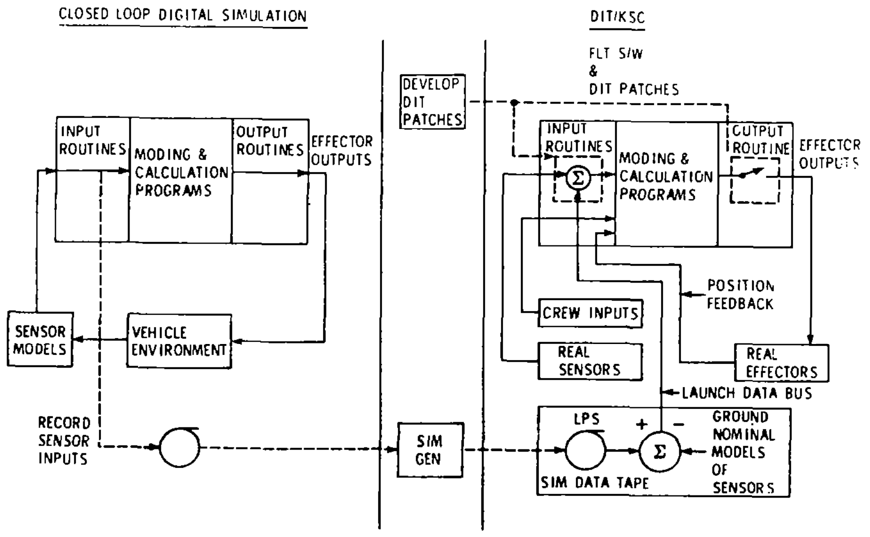


Figure 2

DIT DEVELOPMENT SCHEDULE

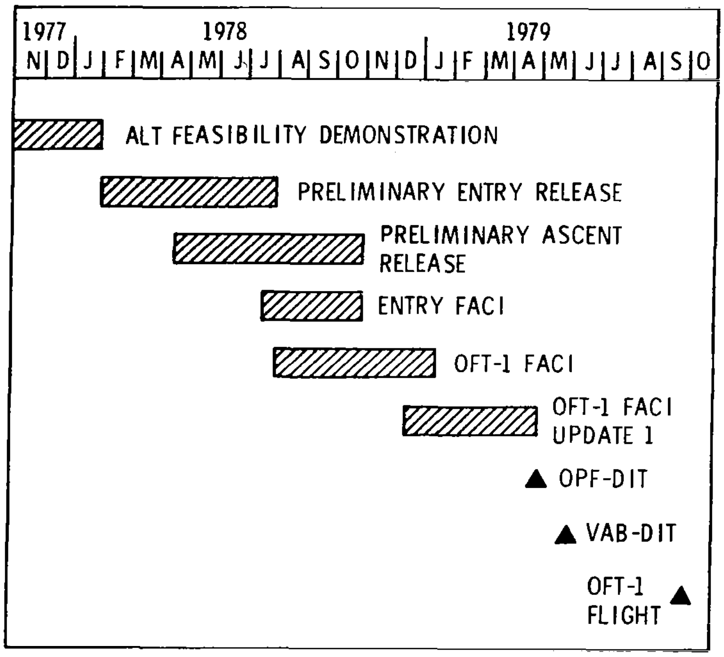


Figure 3

LAB CONFIGURATION FOR DIT TESTING

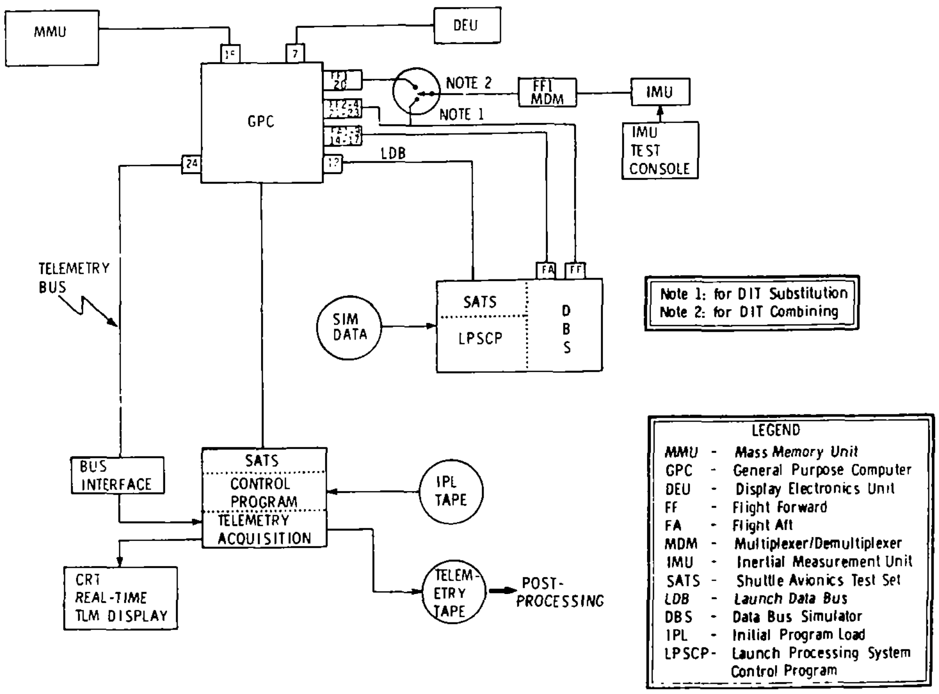
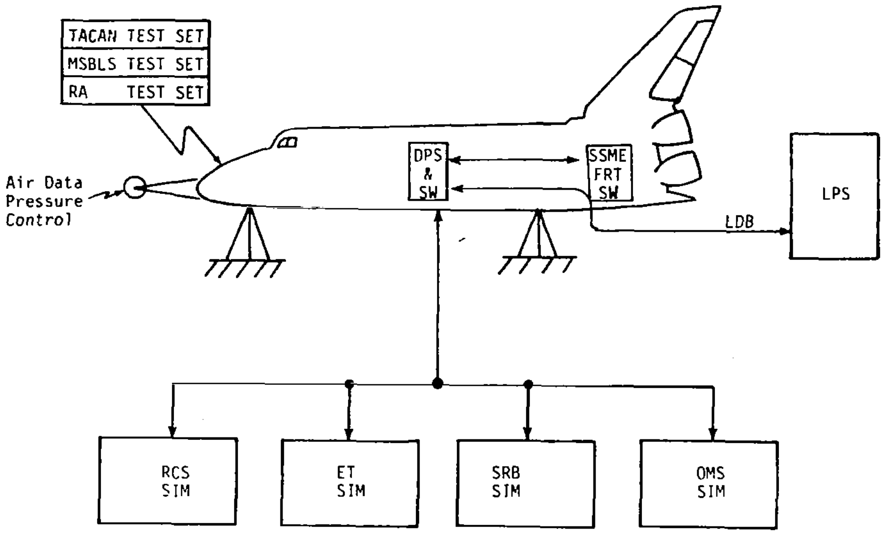

MMU
DEU
GPC
7
24
20
LDB
NOTE 2
NOTE 1
FFI MDM
IMU
IMU TEST CONSOLE
FA
FF
TELEMETRY BUS
SIM DATA
SATS
LPSCP
D B S
Note 1: for DIT Substitution
Note 2: for DIT Combining
BUS INTERFACE
SATS CONTROL PROGRAM
TELEMETRY ACQUISITION
IPL TAPE
CRT REAL-TIME TLM DISPLAY
TELEMETRY TAPE
POST-PROCESSING
LEGEND
MMU - Mass Memory Unit
GPC - General Purpose Computer
DEU - Display Electronics Unit
FF - Flight Forward
FA - Flight Aft
MDM - Multiplexer/Demultiplexer
IMU - Inertial Measurement Unit
SATS - Shuttle Avionics Test Set
LDB - Launch Data Bus
DBS - Data Bus Simulator
IPL - Initial Program Load
LPSCP - Launch Processing System Control Program

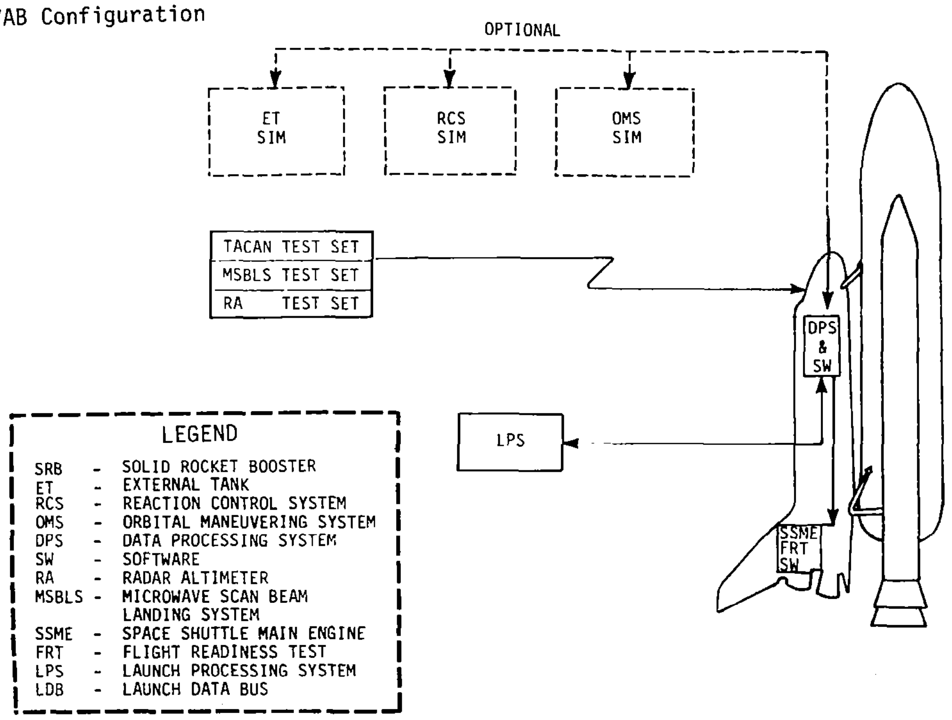
Figure 4



OPF Configuration

Figure 5

VAB Configuration



SRB	-	SOLID ROCKET BOOSTER
ET	-	EXTERNAL TANK
RCS	-	REACTION CONTROL SYSTEM
OMS	-	ORBITAL MANEUVERING SYSTEM
DPS	-	DATA PROCESSING SYSTEM
SW	-	SOFTWARE
RA	-	RADAR ALTIMETER
MSBLS	-	MICROWAVE SCAN BEAM LANDING SYSTEM
SSME	-	SPACE SHUTTLE MAIN ENGINE
FRT	-	FLIGHT READINESS TEST
LPS	-	LAUNCH PROCESSING SYSTEM
LDB	-	LAUNCH DATA BUS

Figure 6

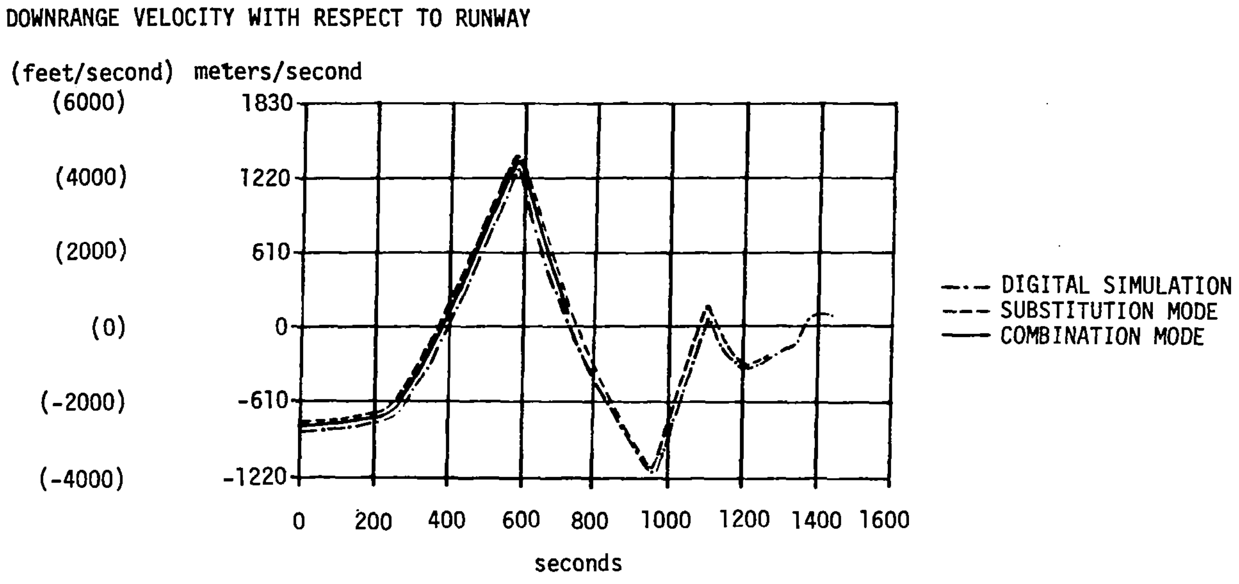


Figure 7

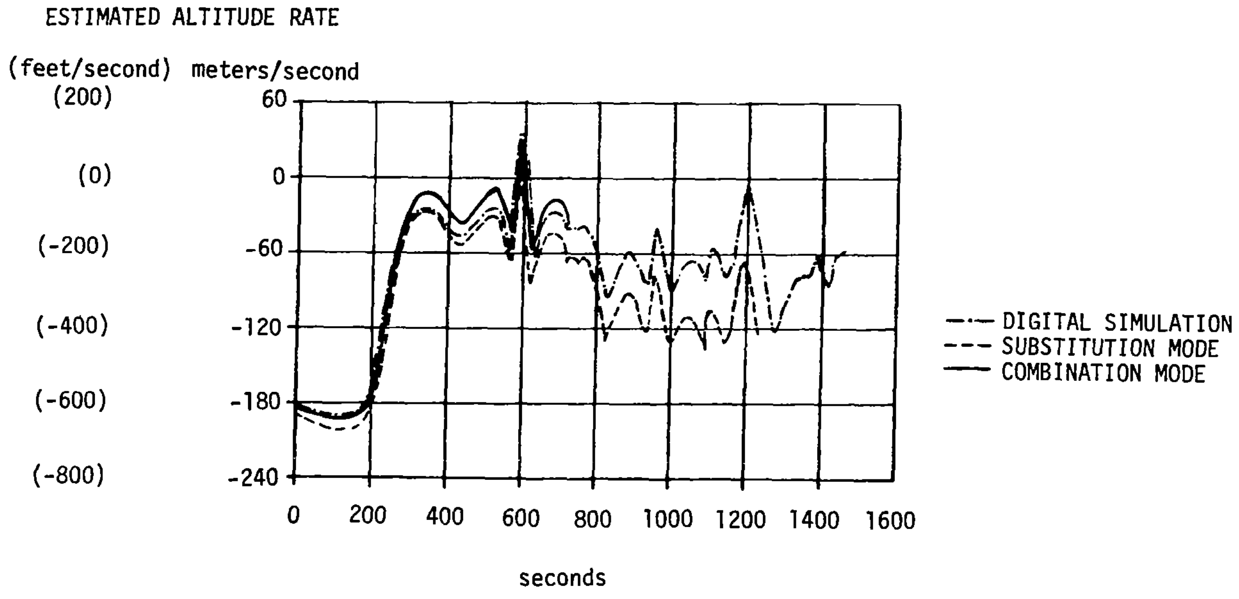


Figure 8

1. Report No. NASA CP-2064	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle TOOLS FOR EMBEDDED COMPUTING SYSTEMS SOFTWARE		5. Report Date November 1978
		6. Performing Organization Code
7. Author(s)		8. Performing Organization Report No. L-12640
		10. Work Unit No. 506-20-13-02
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665		11. Contract or Grant No.
		13. Type of Report and Period Covered Conference Publication
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14. Sponsoring Agency Code
15. Supplementary Notes		

16. Abstract

NASA, in cooperation with the AIAA Computer Systems Technical Committee, sponsored the "NASA/AIAA Workshop on Tools for Embedded Computing Systems Software" in Hampton, Virginia, on November 7-8, 1978. The objectives of the workshop were to assess the current state of tools for embedded systems software and determine future directions for tool development. A synopsis of the talk and the key figures of each workshop presentation, together with chairmen summaries, are included in this publication. The presentations covered four major areas: Tools and the Software Environment (development and testing); Tools and Software Requirements, Design, and Specification; Tools and Language Processors; and Tools and Verification and Validation (analysis and testing). The presentations described described and assessed the utility and contribution of existing tools and recent research results for the development and testing of embedded computing systems software.

17. Key Words (Suggested by Author(s)) Embedded computing systems software; Software tools; Software development and testing environment; Software requirements; Design and specification tools; Language processing tools; Software verification and validation tools	18. Distribution Statement Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages 140	22. Price* $7.25

* For sale by the National Technical Information Service, Springfield, Virginia 22161