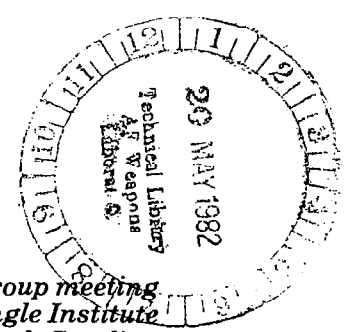
NASA Conference Publication 2222

Production of Reliable Flight-Crucial Software

Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting



Proceedings of a sub-working-group meeting held at Research Triangle Institute Research Triangle Park, North Carolina November 2-4, 1981

NASA

NASA Conference Publication 2222

Production of Reliable Flight-Crucial Software

Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting

Edited by
J. R. Dunham
Research Triangle Institute

J. C. Knight
University of Virginia

Proceedings of a sub-working-group meeting
held at Research Triangle Institute
Research Triangle Park, North Carolina
November 2-4, 1981

NASA
National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1982

PREFACE

As a part of an on-going reliability validation research program, NASA Langley Research Center sponsored a Sub-Working-Group Meeting on the Production of Reliable Flight-Crucial Software. This meeting, which was held at Research Triangle Institute, November 2-4, 1981, specifically addressed the state of the art in the production of crucial software for flight control applications. It provided a forum where researchers communicated their ideas about how to develop highly reliable software and highlighted problems associated with reliable software production.

Meeting objectives were to survey the state of the art and identify areas where additional research is needed. A more specific objective of the sub-working-group meeting was to obtain answers to the following questions:

1. Is it meaningful to associate reliability metrics with software? If so, what are these metrics and how are they to be computed?
2. How good are the classical methods used in the conventional software development cycle? Are they adequate for building crucial software assuming a composite set of quality metrics was defined?
3. Are the more modern formal methods of building software sufficiently mature that they could be applied during the production of reliable software for digital flight control systems?

The consensus was that it is meaningful to associate reliability metrics with software. However, the precise nature of these metrics needs to be determined.

Classical methods are inadequate for achieving a failure probability of 10^{-9} for a 10-hour flight. It was suggested that employing an eclectic set of complementary techniques constitutes a feasible near-term solution using classical methods. This approach should yield a substantial improvement in the reliability of a given software system.

Some formal methods are approaching feasibility for production use. Technical advances in the manageability of these methods must occur prior to their adoption.

The meeting format involved brief and informal presentations followed by discussion. The earlier sessions considered conventional approaches to reliable software development while the later ones focused more on reliability measurement and the more formal methods. All presentations addressed the state of the art of the methodology under consideration. A general discussion of the main problems and research needs was held in the latter part of the second day.

Each meeting participant submitted a prioritized list of three short-term and three long-term research needs. The results of this prioritization activity indicated a short-term need for research in the areas of tool development and software fault tolerance. For the long term, research in formal verification or proof methods was recommended. Formal specification and software reliability modeling were recommended as topics for both short- and long-term research. Recommendations for research include the use of the NASA Avionics Integration Research Laboratory (AIRLAB).

This sub-working-group meeting on the production of reliable software was conceived and sponsored by personnel at NASA Langley Research Center, in particular Billy L. Dove and A. O. Lupton.

CONTENTS

1.0 INTRODUCTION AND OVERVIEW

1.1 Problem Motivation

A computer application is termed crucial if failure could endanger human life. An example is a full-time, full authority digital flight control system for commercial air transport. Present commercial aircraft use mechanical and hydraulic linkages and analog controls in flight-critical applications. The next generation of aircraft is expected to use digital flight controls and digital communications between the control system and the control surface actuators. Research in this area is important since there is the potential for substantial fuel savings and improved aircraft performance associated with entirely digital control systems.

Crucial applications demand high reliability as well as validation that the reliability prediction is meaningful. A requirement of a system failure probability of 10^{-9} for a 10-hour flight has been used as a working figure. This reliability requirement is a system requirement and therefore includes system failures resulting from either hardware or software anomalies. A great deal of work has been done on systems designed to be tolerant of hardware faults [1,2]. However, further work is needed to determine how to measure, cope with, or eliminate faults which occur in software.

The problem of software quality has been studied extensively, but usually with the imprecise goal of improving quality rather than achieving a certain specified reliability figure. For digital flight control systems to be accepted as suitable for commercial use, it will be necessary to show that the required software reliability has been achieved.

1.2 Meeting Objectives

The meeting objectives included surveying the state of the art in reliable software production and identifying research needs. A more specific objective of the sub-working-group meeting was to obtain answers to the following questions:

1. Is it meaningful to associate reliability metrics with software? If so, what are these metrics and how are they to be computed?
2. How good are the classical methods used in the conventional software development cycle? Are they adequate for building crucial software assuming a set of quality metrics was defined?
3. Are the more formal modern methods of building software sufficiently mature that they could be applied during the production of reliable software for digital flight control systems?

Research recommendations could include the use of the NASA AIRLAB facility.

1.3 State of the Art in the Production of Reliable Software

Present day production of software for crucial systems relies on a balanced allocation of a myriad of resources and represents a costly endeavor. The developers of the software for the Space Shuttle used accepted technology, review boards, and brute force testing to maximize the reliability of the software and their confidence in it. Their goal (which coincides with the goals of the IBM Cleanroom project) was to produce error-free software. Whether error-free software is attainable remains an open question.

Various software engineering approaches exist which contribute to the reliability of a software system. The extent of their contributions still needs to be quantitatively determined.

Formal specifications are a critical aspect of highly reliable software and presuppose a mechanism for determining the equivalence of the specification with the intent. Technology for addressing this problem does not exist today.

The available software reliability models require very large amounts of execution time to produce accurate estimates if the software is close to achieving a failure probability of 10^9 in a 10-hour flight. The problems of assuring the reliability of software may be more difficult than those encountered during attempts to produce it.

The reliability of any software depends on the reliability of a considerable body of support software (tools, languages, processors, etc.). High-level language implementations must be reliable if programs written in those languages are to be reliable. Experimental efforts are under way to collect a set of tools in a unified system for programmers' use. Systems which will assist management with administration of a software project are also under development.

1.4 Summary of Results

The overriding group consensus was that the currently stated reliability requirements for software alone cannot be achieved or confirmed with current technology. Available evidence indicates that current reliability figures are orders of magnitude less than required.

For the short term the highest priority research recommendations were:

1. formal specification
2. software environment and tool development
3. reliability prediction, estimates, and measurement
4. fault-tolerant designs
5. formal verification

For the long term the highest priority research recommendations were:

1. reliability prediction, estimation, and measurement
2. formal specification
3. formal verification

It was suggested that AIRLAB might serve as a repository for sample flight control problems, support tools, and experimental results. Statistically controlled software development experiments using flight control problems as vehicles for coherency could be performed in AIRLAB. These experiments would permit measurement of the contributions that different software development methodologies make to reliability.

2.0 RELIABLE SOFTWARE DEVELOPMENT PROJECTS

2.1 Producing Reliable Software for the Space Shuttle

The Space Shuttle Software System is unique in that it uses software to perform crucial functions with no analog backup. The primary goals of the Shuttle software developing agency (IBM-FSD) were to produce software which meets the intent of customer requirements, have the software perform in accordance with the customer's operational expectations, and produce software which is free from errors. The use of a composite set of midseventies techniques comprised the software development process. Software reliability measures, formal specification languages, and formal verification methods were not used.

Four system test facilities were used throughout the production process. These are: a) a software development laboratory, b) a software-hardware integration laboratory, c) a flight systems laboratory, and d) a crew training laboratory. The software developers placed greater emphasis on the earlier part of the development cycle. An early definition of development tools, the use of structured methodologies, strict configuration control, and the extensive use of review boards for decision making constituted the development approach. The developers fostered an adversary relationship between the designers and verifiers by maintaining their organizational independence. A concise yet thorough description of the Shuttle software development is given in a paper by A.J. Macina [3].

One of the difficulties encountered during the development of the Shuttle software was the need to overlay software programs in memory. The function and size of the applications software were not considered in the hardware selection decision. In retrospect, it seems that software size should weight this decision.

The selection of the quad-redundant design also posed problems in that a 2 by 2 split was possible. Considerable effort was allocated to assuring that this condition did not occur. In the development of the SIFT and FTMP computers this problem was solved via the theory of interactive consistency [4].

Since reliability measures were not produced, the Shuttle developers have no measure of the reliability achieved. Brute force testing has increased their confidence but in an unquantified way. In addition, they are only minimally confident of correct operation in off-nominal flight operation. The software failed during simulation of an off-nominal situation 3 weeks prior to the second Shuttle launch.

2.2 Controlling the Software Development Process - The SAGA Project

SAGA, a syntax-directed management system for software production, is an ongoing research project at the University of Illinois [5]. This effort is aimed at tackling the complexity of software development projects by providing an interactive system for formally describing software production management and controlling the mechanization of management policies.

Context-free grammars, called management grammars, which describe the software development process have been proposed, and recognizers for such grammars are currently being developed. In use, the recognizers will permit only approved activities by software project staff and will collect management data routinely and automatically as the project proceeds.

The goals of the SAGA project represent an aggressive effort towards making the process of software development more visible. This project illuminates the need to use computers to control the development of computer programs. If successful, the project will enable management to make decisions based upon accurate up-to-date information and to better control the software development process.

2.3 The Cleanroom Approach to Reliable Software Development

The IBM Cleanroom Software Development Project constitutes a technical and organizational approach to developing software products with certifiable reliability. This approach divides software development into two parts: software design engineering and software product engineering. The design engineer creates the product and the product engineer certifies it.

The methods employed by the design engineers include stepwise refinement, correctness proving, finite state machine definitions, and the use of a design language. In the coding phase, the design engineers use high-level programming languages, structured programming techniques, and code reviews. They are trained to have the attitude that they can produce error-free software and are permitted to perform only syntax checks on their code.

The product engineers essentially debug the software produced by the design engineers. The strategy used by the product engineers involves blind testing in which design details are hidden. This testing is accomplished by analyzing the input probability distributions, generating random inputs according to these distributions, and recording failure data. Mean-time-before-failure statistics are generated, and the product's reliability is estimated using Musa's execution time model [6]. Regression testing occurs as part of the failure diagnostic support.

Two premises underlie the Cleanroom approach to developing highly reliable software. One premise is that individuals can be taught to write correct programs. Arguments which support and question this assumption can be constructed. By removing the crutch of testing, the designers will most probably be more conscientious in their code development and more apt to subject their code to extensive desk checking. On the other hand, are humans actually capable of consistently writing error-free code?

The second premise is that randomized testing by itself is sufficient. Randomized testing definitely avoids the 'fix the bug' and 'intended use' syndromes which designers are prone to exhibit during testing. On the other hand, path testing and tests which detect error types that occur most frequently are far from useless. Furthermore, a difficulty encountered in randomized testing is the inability to predict the correct output.

2.4 Preimplementation Phases of Software Development

A range of approaches [7] exists for specifying the preimplementation information needed in the initial phases of software development. These specification approaches include both the informal traditional and information flow methods and the more formal state-based, expression-based, axiomatic, and temporal logic descriptions. Coupled with the choice of specification language are the types of analysis that it is desirable to perform. Checking the consistency of the way the information is used represents an extant analytical technique. Rapid prototyping, simulation, and testing constitute viable yet infant approaches to ensuring the correctness of formal language specifications. Proofs methods may be useful for verifying specifications written in axiomatic description languages.

A clear delineation between the types of information recorded during each of the preimplementation phases of software development does not exist. For example, the distinction between requirements and design is often vague. The boundaries of and transition between each of these phases must be precisely defined. This definition is prerequisite to the selection of a description language. The choice of a description language should depend upon the amount and type of information to be specified during each preimplementation phase.

For flight control systems, a language suitable for describing the desired concurrency, real-time constraints, and response to exceptions is needed. Research on the types of analysis necessary to establish the completeness and correctness of flight control software requirements is also needed.

2.5 Programming Languages

Requirements for high-level programming languages include power of expression and reliability. ADA, EUCLID, and GYPSY are examples of high-level programming languages developed for producing reliable software. Features of these languages include strong data typing, the use of data and procedure abstractions, exception handling facilities, and the ability to express concurrency. Strong data typing permits the compile-time checking of the consistent use of variables. Data abstractions and procedure abstractions are mechanisms for selectively hiding objects and allowing partial access. The inclusion of precertified packaged routines and library facilities also enhances the reliability of programs written in these languages.

Although they may offer the ability to write more reliable programs, these languages do not guarantee the production of software having a reliability of the order necessary because many of the traditional sources of error remain possible. In addition, their language implementations have not been certified as reliable. Proving the correctness of an entire implementation for a language like ADA is beyond the current state of the art.

One of the difficulties underlying the formal verification of an entire language implementation stems from the lack of adequate methods for defining the semantics of programming languages, and the considerable body of support software needed in addition to a compiler. Except for the control flow constructs, the run-time structure of a program differs from the static compile-time structure. Demonstrating that a program is ultra-reliable will require knowledge of its run-time structure and hence knowledge of the compiler's implementation (i.e. the language semantics). This demonstration is referred to as proof of security of implementation. Verification based on models of programs as they are executed requires additional research. Proofs of implementation that include all support software as well as the compiler also require extensive additional research [8,9].

2.6 Software Testing

One method for constructing test data sets which yield increased confidence in program correctness involves evaluating test data sets by introducing errors into a program P. This method is known as program mutation [10]. Program mutation consists of constructing a test data set, executing P with the test data, introducing errors into P to form a mutant P', executing P' using the original test data, and comparing the results to see if the test data distinguished P from P'. The number of program mutations constructed is reduced by assuming that the programmers are competent and will try to deliver a correct program. For example, mutating a program by deleting it entirely is a valid mutation but is clearly pointless.

Metrics can be calculated for various test data sets. One possible metric is the percent of nonequivalent mutants of P which were distinguished by the test data. The usefulness of this metric lies in its ease of computation.

This methodology evaluates the effectiveness of test data sets in detecting various types of simple errors. Program failures resulting from incomplete specifications or missed requirements are excluded. Some data exist which indicate that the majority of complex errors are comprised of combinations of simple errors. This phenomenon implies that attaining a high degree of test coverage for simple errors will provide some coverage of the more complex errors. Further characterization of the error space is needed.

The mutation method of program testing represents one approach to evaluating test data sets. Other approaches exist in the literature [11]. A study which evaluates the efficiency of the various testing approaches constitutes a valid research need.

2.7 Software Fault Tolerance

Software fault tolerance methods are methods for developing software which is tolerant of software faults [12]. A software fault is defined as a design defect in the software, where the term "design defect" encompasses all deficiencies introduced throughout the software development process. Manifestation of a software fault places the system in an erroneous state, which may lead to system failure. Recovery blocks, n-version programming, and robust data structures are fault-tolerant mechanisms advocated in the literature today.

Software fault tolerance methods are needed because fault avoidance and fault removal methods alone are inadequate for achieving the required level of reliability. Implemented in unison, fault tolerance, avoidance, and removal represent a balanced approach to producing highly reliable software.

Experiments which assess the contributions to reliability of the various software fault tolerance approaches are needed.

2.8 Software for Flight Control Applications

Producing flight control systems whose reliability is demonstrable requires much effort and expense. The difficulties encountered lie more with the software than the hardware since a suitable framework does not yet exist for quantitatively assessing software reliability. Generally speaking, software reliability is discontinuous. Software failures occur as results of random encounters with design faults rather than results of continuous degradation or wearing out.

The successful handling of software errors involves minimizing the likelihood of error introduction, improving the effectiveness of methods for detecting hidden design faults, and a priori code stabilization. The use of constructive software development methodologies helps minimize the number of errors introduced. Tiger team inspection represents a suitable means for detecting latent faults. A tiger team is a sophisticated group of individuals who function in a constructive yet adversary manner. Crucial software (e.g. the executive) may be stabilized by extensive use in noncritical applications. Stabilized code could then be placed in libraries for multiapplication use..

Since software failures frequently cause the system to exhibit aberrant and discontinuous behavior, it may be advantageous to invoke a procedure which results in a seemingly continuous system recovery. This recovery may be accomplished by reinitializing the system to a previously correct state. Note that the effects of repeated invocation of a reinitialization procedure can be observed easily [13].

2.9 Software Environments - The TOOLPACK Project

A software environment provides programmers with an integrated set of tools which assist them in creating software [14]. The TOOLPACK project is an ongoing endeavor to establish the appropriate environment for FORTRAN programmers who write small- to medium-sized mathematical programs.

Encouraging programmers to experience the tools and provide feedback is a basic tenet of the TOOLPACK project. To initiate a feedback loop, the project leaders have designed small experiments aimed specifically at identifying the proper tools, information base, and user interface. Current components of this integrated system are tools which support code development, maintenance, testing, analysis, documentation, and portability. A variety of institutions which develop mathematical software are participating in this project.

The data files are an essential component of the TOOLPACK system. Since the data files are implemented by the host's file system, portability is enhanced by utilizing a very simple and commonly occurring interface. It is difficult to determine a priori what the organization and contents of these files should be. An intelligent guess must be made. Once knowledge of usage patterns increases, an accordant information base will be constructed.

Gaining acceptance by the user community represents a major obstacle for TOOLPACK. The incremental development approach, the flexibility provisions (i.e. no predefined order of tool uses), and an intelligent editing facility should contribute to its acceptance. A remaining critical problem, however, is the response time required. An obvious solution might be an overnight run which establishes the initial information base.

2.10 Static Analysis of Concurrency

A tool which eventually will be incorporated into an integrated environment is being developed to statically analyze concurrency in ADA programs. The specific analytical capabilities being developed document rendezvous, parallel actions of interest, and potential deadlock or infinite wait situations. This concurrency analyzer requires the user to have some knowledge of the underlying analysis. In particular, the user must be capable of resolving problems which surface during the analysis.

For languages which permit rendezvous, analyzing concurrency and detecting erroneous conditions is np-complete. The computation time required for detecting erroneous situations grows exponentially with n, the number of concurrent tasks being statically analyzed. This analysis is manageable for $n \leq 5$.

Heuristics may be developed to allow analysis of larger systems of tasks. However, work is proceeding on an algorithm to reduce the complexity of systems involving larger numbers of tasks by partitioning the set and analyzing each set independently [15,16].

2.11 Software Reliability Measurement

An important issue in crucial software development is the quantification of its reliability. Problems encountered in assessing the reliability of flight software may prove more difficult than those encountered during attempts to produce it. Reviews of software reliability models can be found in the literature [17,18,19]. Assuming that these models are applicable to the systems of interest, they will still require a very large amount of execution time in order to estimate reliability when it is of the order required.

A potential solution is to develop a model which investigates the internal structure of the software. In developing such a model, reliability theorems for hardware designs may prove useful. A theorem from hardware reliability states that redundancy at the component level results in a more reliable system than redundancy at the system level. This theorem may be useful for assessing the contributions that the various fault-tolerant methods make to reliability. It remains to be investigated whether or not this theorem holds true for software.

The modeling of software reliability is extremely important to the development of acceptable crucial systems. It is not clear, however, whether the existing models of reliability are appropriate to digital flight control system software. In addition, these models have not been extensively validated by experiment. A great deal of research is needed before believable reliability figures can be associated with software.

2.12 Formal Verification of SIFT

Formally verifying the design of SIFT entailed the specification of a hierarchy of models [20]. The highest level model within this hierarchy is the I/O model. This model succinctly describes the required properties of the system. To gain assurance in the correctness of the SIFT design, the policy maker needs to understand only the description of the axioms of this top level model. In SIFT, there are six such axiomatic statements. The lowest level model in the SIFT hierarchy describes the program executed by the hardware.

The models of the system are specified using different languages. Axioms in one model are mapped to axioms specified in the next level model. Given these mappings, verification of the design of SIFT involves showing that each axiom specified in a higher level model is provable as a theorem of a lower level model. This methodology alleviates the dangers of inconsistency by providing assurance that one axiom is derivable from the next.

The proof of correctness of the SIFT software is a little over 100 pages. This does not include the lowest of the six levels. The majority of difficulties in the proof technique have been eliminated and it is anticipated that an intelligent computer science graduate could prove the software correct in about 6 months.

Essential to using a semiautomated theorem prover is the creation of a symbiotic relationship between man and machine. The human understands the proof and can use intuition to choose between the numerous paths which could be taken. The computer system is useful for simplification and provides the required bookkeeping services. Critical aspects of this symbiosis are short response times between the steps of the proof and adequate information display facilities. If the response time is too long, the human is forced to specify greater detail and take smaller steps in directing the system. Proving SIFT required approximately 18 megabits of store. Approximately

500 lemmas were created, which posed a considerable bookkeeping problem. Part of the problem was the difficulty of maintaining meaningful lemmas on a single screen.

The construction of the SIFT design proof was pedagogical. It demonstrated the feasibility of design proofs and highlighted the importance of the man-machine symbiosis. It also reinforced the need for simplicity. Violation of simplicity may present undue restrictions on what can be assuredly demonstrated about a system. Further work which will enhance the formal verification process is needed.

2.13 System Specification and Program Transformation

A trend in reliable software development involves the definition of new ways in which support software can be used to effectively eliminate sources of error. Discrepancies between the user's intent and the actual system specification constitute a major source of errors. Transitions between steps in the system life cycle are another potential source of errors. Software which assists in the formulation of system specifications and semiautomates program transformations should enhance the reliability of a system by eliminating these error sources. Developing this support software is not an easy feat.

Specification Acquisition from Experts (SAFE) is a prototype tool which simplifies the creation of a formal specification [21]. The development of this tool is highly desirable as it should increase the reliability of the specification process and does not require specialized training. It would also make the formal specification more maintainable, since the informal specification can be modified and semiautomatically retransformed into the formal specification. Making the SAFE system interactive helps eliminate the problem of computer misinterpretation of the informal specification. It has been demonstrated that this interaction can be kept to a minimum so as not to abrogate the advantages of informality. Part of this project involved the development of a suitable formal specification language.

Developing computer-based tools which support the user during the development of a program by mechanically transforming formal specifications into efficient implementations should improve reliability. If the transformation correctly preserves semantics, as intended, new errors cannot be introduced. A prototype program transformation tool is currently under development at USC-ISI [22]. This tool transforms specifications using a methodology similar to that used for verifying SIFT. The main difference is that program optimization and maintainability rather than verification are of concern. Maintainability is improved since changes and enhancements are effected by modifying the specification and allowing the computer to redo the transformations which resulted in the original optimized implementation. Since optimizing a program contributes to its complexity and hence its reliability, this automated approach seems superior to conventional maintenance.

These tools require additional development before they will be useful for production. Since the man-machine interface is extremely important, some trial use and feedback as with the TOOLPACK project will be necessary. Note that these types of automation would allow one person to create a software system. The problem has not been tackled for permitting development by a group of people. An effective communication mechanism would have to be developed in this case.

3.0 CONCLUSIONS

The overriding group consensus was that the currently stated reliability requirements for software alone cannot be achieved or confirmed with current technology. Available evidence indicates that current reliability figures are orders of magnitude less than required.

A concern which was voiced repeatedly by working group participants was the need for improved methods of defining requirements and specifications. Experience has shown that inconsistent or inadequate requirements definitions are a constant source of errors which are exceedingly difficult to find.

In the short term the best overall approach to software development is to employ an eclectic set of complementary techniques. The integration of fault avoidance, fault tolerance, fault removal, and other software engineering methods should yield a substantial improvement in overall reliability. Component technology at all levels of software development is also recommended as the components may be separately validated and reused.

In the long term, formal modeling and definition methods should prove invaluable at all levels of software development for the projected reliability requirements. Techniques which investigate program structure should prove immensely useful. Demonstrating absolute equivalence between a specification and an implementation may be technically attainable. There will remain, however, the difficulty of achieving the absolute equivalence of the specifications with the intent.

The following is a list of the general comments agreed upon by the majority of the meeting participants during the discussion period held at the latter part of the second day. They are not prioritized.

Formal specifications are a critical aspect of super reliable software and presuppose a mechanism for determining the equivalent of the specification with the intent. We do not have technology for addressing that problem yet.

The currently stated reliability requirements for software alone cannot be confirmed with current technology. All available evidence indicates that we are currently several orders of magnitude short of the stated figure in general. It is unlikely that we can achieve ultra reliability by incremental improvements in reliability.

There is serious doubt that it is presently possible to produce flight software systems having the stated level of reliability and to assure that they have that level of reliability.

We do not have a measure of the level of reliability that can be assured by a methodology, or the ability to compare the levels assured by different methodologies.

The reliability of any flight software depends on the reliability of the considerable body of support software (tools, language processors, etc.).

Within the forseeable future it will not be possible to define highly reliable requirements of an arbitrarily complex system. We must learn to limit the complexity of systems or at least of those parts that must be reliable.

"In the short term, the best approach is to be eclectic. We recommend an integration of fault avoidance, fault tolerance, etc. and we expect a substantial improvement. We don't know what the right mix is!"

"In the longer term, absolute equivalence between a specification and an implementation may be technically attainable.

"Formal modeling and definition methods are invaluable at all levels of software development for the projected reliability requirements. Make them comprehensible, concise and intellectually manageable by mere mortals."

"We recommend component technology at all levels of development. These can be separately validated and reused."

"The contributions which the various software engineering approaches make to reliability need to be quantitatively determined."

"We are dismayed that in the area of hardware reliability little or no attention is given to modeling and analyzing design faults. These faults are similar to software faults, and are the source of most system unreliability."

"Do what we already know in real applications."

"The internal structure of the software cannot be ignored."

Due to the scope and inherent complexity of the problem being addressed, a prioritization of research needs was requested. This prioritization was accomplished by a vote in which each meeting participant ranked three short-term and three long-term research needs. Table 1 summarizes the results of this vote and shows formal specification, software environments/ tools, reliability modeling, fault-tolerant designs, and formal verification as the foremost short-term research needs. Research in reliability modeling, formal specification, and formal verification is indicated for the long term. These recommendations are listed by participant in the appendix.

Recommendations for research included suggestions that AIRLAB be a repository for sample flight control problems of various sizes. These problems would be useful for quantitatively evaluating the various approaches to reliable software production. This evaluation could take the form of statistically controlled software development experiments. These experiments would involve the fabrication of software solutions for real applications and require all activities from the informal problem statement to the highly reliable software product. Thus, these experiments would be termed "end-to-end." An experimenter might deliver a prepackaged system component and the overall development process could be evaluated within AIRLAB. Thus, AIRLAB could serve as a repository of sample problems, system development tools, and experimental results. It could be a place where comparative and competitive studies are performed as well as a focal point for additional workshops.

The following is a list of AIRLAB themes recommended by the meeting participants:

- Repository of sample avionics problems, various sizes
- Tool and result repository
- University experiments in all areas

- Coordination and integration of results
- Comparative, competitive studies
- Additional workshops
- Collection of statistically meaningful data

4.0 REFERENCES

1. Hopkins, A. L.; Smith, T. Basil, III; and Jaynarayan, H. Lala: FTMP: A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. Proceedings of the IEEE, vol. 66, no. 10, October 1978, pp. 1221-1239.

2. Wensley, John H.; Lamport, Leslie; Goldberg, Jack; Green, Milton W.; Levitt, Karl N.; Melliar-Smith, P. M.; Shostak, Robert E.; and Weinstock, Charles B.: SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. Proceedings of the IEEE, vol. 66, no. 10, October 1978, pp. 1240-1255.

3. Macina, A. J.: Independent Verification and Validation Testing of the Space Shuttle Primary Flight Software System. Paper presented at NSIA/AIA/USAF-SD/NASA Conference and Workshops on Mission Assurance, Los Angeles, Ca., April 1980.

4. Pease, M.; Shostak, R.; and Lamport, L.: Reaching Agreement in the Presence of Faults. J. of the ACM, vol. 27, no. 2, April 1980, pp. 228-234.

5. Campbell, R. H.; and Richards, P. G.: SAGA - A System to Automate the Management of Software Production. AFIPS Conference Proceedings: 1981 National Computer Conference, AFIPS Press, Arlington, Va., 1981, p. 231.

6. Musa, J. D.: Validity of the Execution Time Theory of Software Reliability. IEEE Transactions on Reliability, vol. R-28, no. 3, August 1979, pp. 181-191.

7. IEEE Transactions on Software Engineering, vol. SE-3, no. 1, Jan. 1977, pp. 1-102.

8. Pratt, T. W.: H-Graph Semantics - Data Structure Grammars. Report no. 81-15, Dept. of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, October 1981.

9. Pratt, T. W.: H-Graph Semantics - H-Graph Machines. Report no. 81-16, Dept. of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, October 1981.

10. DeMillo, Richard A.; Lipton, Richard J.; and Sayward, Frederick G.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer, vol. 11, no. 4, April 1978, pp. 34-41.

11. Howden, W. E.: A Survey of Dynamic Analysis Methods. IEEE Tutorial: Software Testing and Validation Techniques, EH0138-8, September 1978, pp. 184-206.

12. Anderson, T. A.; and Lee, P. A.: Fault Tolerance: Principles and Practice. London: Prentice-Hall Intl., Inc., 1981.

13. Schwartz, Jacob T.: Comments on Highly Reliable Software for Avionics Applications. ICASE rep. no. 81-31 (Contracts NAS1-14472 and NAS1-15810), Univ. Space Res. Assoc., Sept. 23, 1981. (Available as NASA CR-165878.)

14. Osterweil, L.: Software Environment Research: Directions for the Next Five Years. Computer, vol. 14, no. 4, April 1981, pp. 35-43.

15. Taylor, R.: An Algorithm for Analyzing Concurrent Programs. Report no. DCS-10-IR, Dept. of Computer Science, University of Victoria, British Columbia, Canada, May 1981.

16. Taylor, R.: Complexity of Analyzing the Synchronization Structure of Concurrent Programs. Report no. DCS-10-IR, Dept. of Computer Science, University of Victoria, British Columbia, Canada, May 1981.

17. Littlewood, B.: Theories of Software Reliability: How Good Are They and How Can They Be Improved? IEEE Transactions on Software Engineering, vol. SE-6, no. 5, September 1980, pp. 489-500.

18. Musa, J. D.: The Measurement and Management of Software Reliability. Proceedings of the IEEE, vol. 68, no. 9, September 1980, pp. 1131-1143.

19. Schick, G. J.; and Wolverton, R. W.: An Analysis of Competing Software Reliability Models. IEEE Transactions on Software Engineering, vol. SE-4, no. 2, March 1978, pp. 104-120.

20. Melliar-Smith, P. M.; and Schwartz, Richard L.: Heirarchical Specification of the SIFT Fault-Tolerant Flight Control System. SRI Technical Report No. CSL-123, SRI International, Menlo Park, Ca., March 1981.

21. Balzer, R.; Goldman, Neil, and Wile, David.: Informality in Program Specifications. IEEE Transactions on Software Engineering, vol. SE-4, no. 2, March 1978, pp. 94-106.

22. Balzer, R.: Transformational Implementation: An Example. IEEE Transactions on Software Engineering, vol. SE-7, no. 1, January 1981, pp. 3-13.

Table 1. Recommended Research Priorities

Activity	Rank Sum	
	Short Term	Long Term
Formal Specification	14	14
Formal Verification	9	12
Reliability Prediction, Estimation & Measurement	12	15
Programming Language	2	2
Software Environments/ Tools	14	8
Test Methods	4	2
Fault-Tolerant Designs	10	0
Real-Time Issues	1	0
Validation Tools & Techniques	6	5
Automatic Programming	0	7
Flight Control Program Libraries	0	4
Management Policies	1	0

APPENDIX

PRIORITIZATION OF RECOMMENDED RESEARCH ACTIVITIES

PARTICIPANT - T. ANDERSON

Short Term

1. Reliability measurements & requirements/specification (tie)
2. Software fault tolerance
3. Real-time issues

Long Term

1. Validation
2. More requirements
3. Development tools

PARTICIPANT - R. CAMPBELL

Short Term

1. Reliability - how to measure - actual ways to measure
2. Formal verification and validation of complete software life cycle including requirements, maintenance and testing
3. Tools to aid in measuring reliability and formal verification and validation of complete software life cycle including requirements, maintenance and testing

Long Term

1. Actual ways to measure reliability
2. Formal verification and validation of complete software life cycle including requirements, maintenance and testing plus much more integration to form a "product"
3. Tools to aid in measuring reliability and formal verification and validation of complete software life cycle including requirements, maintenance and testing plus much more integration to form a "product"

PARTICIPANT - F. DONAGHE

Short Term

1. Specifications
2. Implementation
3. Verification

Long Term

1. Specifications
2. Implementation
3. Verification

PARTICIPANT - M. DYER

Short Term

1. SIFT type verification
2. Fault tolerance (see Anderson)
3. Reliability measures

Long Term

1. Specification methods
2. Spanning specifications to implementation
3. Packaging for components

PARTICIPANT - B. LITTLEWOOD

Short Term

1. Stochastic reliability modeling of software fault-tolerant systems
2. Comparison of performance of existing (and future ?) software reliability models on real data sets
3. Requirements/specifications fault tolerance

Long Term

1. Relationship between other metrics (and quality of them) and software reliability
2. Comparison of subjective beliefs and actual performance; consensus techniques between "expert" witnesses

PARTICIPANT - M. MELLIAR-SMITH

Short Term

1. Formal verification
2. Formal requirements
3. Testing

Long Term

1. Formal verification
2. Formal requirements
3. Software reliability measurement

PARTICIPANT - H. MILLS

Short Term

1. AIRLAB environment for end-to-end model projects
2. Techniques for formal and readable flight software specifications
3. Technical standards for high reliability software development

Long Term

1. Specifications of module (package) library for flight software reuse
2. Automatic programming methods peculiar to flight software
3. Relation of catastrophic theory to to software requirements and specifications

PARTICIPANT - T. PRATT

Short Term

1. Formal methods for specifying and verifying the correctness and consistency of precoding stages of software development requirements, specifications, design
2. Integrated software development environments

   - management tools and software
   - construction/analysis tools

3. Testing methods and quantifying the reliability of software after testing

Long Term

1. Formal methods for specifying and verifying the correctness and consistency of precoding stages of software development requirements, specifications, design
2. Integrated software development environments

   - management tools and software
   - construction/analysis tools

3. Testing methods and quantifying the reliability of software after testing

PARTICIPANT - R. TAYLOR

Short Term

1. A tool environment incorporating the best available technology of

   Requirements analysis
   Preliminary design
   Detailed design
   Coding tools
   Verification and test tools
   (NTS + preimplementation)

2. Software fault tolerance techniques

3. Software management issues

Long Term

1. Preimplementation tools, requirements foremost

2. Program transformation tools

3. Formal verification & Reliability assessment (tie)

PARTICIPANT - J. WILEDEN

Short Term

1. Specifications/requirements tools, especially assessment/animation, for flight control software

2. Contribution of software fault-tolerance to flight control software reliability

3. Contribution of testing to flight control software reliability

Long Term

1. Apropriate methods for measuring reliability and appropriate goals

2. Formal verification

3. Transformation (computer-assisted) implementation from specifications

AGENDA

Monday, November 2, 1981

Introductory Remarks
J. Clary, Research Triangle Institute
AIRLAB Overview
J. Gault, North Carolina State University
Problem Definition
J. Knight, University of Virginia

Tuesday, November 3, 1981

Meeting Introduction
J. Knight, University of Virginia
Producing Reliable Software for the Space Shuttle
F. Donaghe, IBM-Federal Systems Division
Controlling the Software Development Process (SAGA)
R. Campbell, University of Illinois
The Cleanroom Approach to Reliable Software Development
M. Dyer, IBM-Federal Systems Division
Preimplementation Phases of Software Development
J. Wileden, University of Massachusetts
Programming Languages
T. Pratt, University of Virginia
Software Testing
R. DeMillo, Georgia Institute of Technology
Software Fault Tolerance
T. Anderson, University of Newcastle upon Tyne
Software for Flight Control Applications
J. Schwartz, New York University

Wednesday, November 4, 1981

Software Environments (TOOLPACK)
L. Osterweil, University of Colorado
Static Analysis of Concurrency
R. Taylor, University of Victoria
Software Reliability Measurement
B. Littlewood, City University of London
Formal Verification of SIFT
P. M. Melliar-Smith, SRI International
System Specification and Program Transformation
R. Balzer, University of Southern California/
Information Sciences Institute
Summary Discussion

ATTENDEES

Dr. Tom Anderson
Computing Laboratory
University of Newcastle upon Tyne
Claremont Tower, Claremont Road
Newcastle upon Tyne, England
(0632)-329-233

Dr. Robert Balzer
USC/Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, California 90291
(213) 822-1511

Mr. Jim Clary
Digital Systems Research Programs
Research Triangle Institute
Research Triangle Park, NC 27709
(919) 541-6951

Dr. Roy Campbell
Computer Science Department
University of Illinois
at Urbana-Champaign
Urbana, Illinois 61801
(217) 333-6464

Dr. Richard DeMillo
Computer Science Department
Georgia Institute of Technology
Atlanta, Georgia 30332
(404) 894-3180

Mr. Frank Donaghe
Mgr. Test Guidance, Navigation,
and Flight Control
IBM-Federal Systems Division
1322 Space Park Drive
Houston, Texas 77058
(713) 333-7518

Ms. Janet Dunham
Digital Systems Research Programs
Research Triangle Institute
Research Triangle Park, NC 27709
(919) 541-6562

Mr. Michael Dyer
IBM-Federal Systems Division
18100 Frederick Pike
Gaithersburg, Maryland 20760
(301) 493-1495

Dr. Jim Gault
Department of Electrical Engineering
North Carolina State University
P.O. Box 5275
Raleigh, NC 27650
(919) 737-2376

Dr. John Knight
Dept. of Applied Math & Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22901
(804) 924-7201

Dr. Bev Littlewood
Department of Mathematics
City University of London
London, England
01-253-4399 Ext. 4115

Mr. Mike Melliar-Smith
Stanford Research Institute International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200 Ext. 2336

Mr. Earl Migneault
NASA Langley Research Center
Mail Stop 477
Hampton, Virginia 23665
(804) 827-3681

Dr. Harlan Mills
IBM-Federal Systems Division
18100 Frederick Pike
Gaithersburg, Maryland 20760
(301) 493-1495

Dr. Leon Osterweil
Computer Science Department
University of Colorado at Boulder
Boulder, Colorado 80309
(303) 492-6361

Dr. Terri Pratt
Dept. of Applied Math & Computer Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22901
(804) 924-7201

Ms. Janet Schultz
NASA Langley Research Center
Mail Stop 477
Hampton, Virginia 23665
(804) 827-3681

Dr. Jack Schwartz
Courant Institute of Mathematics
New York University
New York, NY 10012
(212) 460-7100

Dr. Richard Taylor
Computer Science Department
University of Victoria
British Columbia, Canada V8R4K3
(604) 721-7228

Dr. Jack Wileden
Department of Computer
and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
(413) 545-0289

1. Report No. CP-2222	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle	5. Report Date
Production of Reliable Flight-Crucial Software - Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting	May 1982
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
J. R. Dunham and J. C. Knight, Editors	L-15291
	10. Work Unit No.
9. Performing Organization Name and Address	11. Contract or Grant No.
Research Triangle Institute Systems and Measurements Division Research Triangle Park, NC 27709	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Conference Publication
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 505-34-43-05

15. Supplementary Notes

J. R. Dunham: Research Triangle Institute, Research Triangle Park, NC 27709
J. C. Knight: University of Virginia, Charlottesville, VA 22901

16. Abstract On November 2-4, 1981, a Validation Methods Research for Fault-Tolerant Avionics and Controls Systems Sub-Working-Group Meeting was held at the Research Triangle Institute, Research Triangle Park, North Carolina, to address the state of the art in the production of crucial software for flight control applications. A more specific objective of the sub-working-group meeting was to obtain answers to the following questions:

1. Is it meaningful to associate reliability metrics with software? If so, what are these metrics and how are they to be computed?
2. How good are the classical methods used in the conventional software development cycle? Are they adequate for building crucial software assuming a composite set of quality metrics was defined?
3. Are the more modern formal methods of building software sufficiently mature that they could be applied during the production of reliable software for digital flight control systems?

The consensus was that it is meaningful to associate reliability metrics with software. However, the precise nature of these metrics needs to be determined.

Classical methods are inadequate for achieving a failure probability of 10^{-9} for a 10-hour flight. It was suggested that employing an eclectic set of complementary techniques constitutes a feasible near-term solution using classical methods. This approach should yield a substantial improvement in the reliability of a given software system.

Some formal methods are approaching feasibility for production use. Technical advances in the manageability of these methods must occur prior to their adoption.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Reliable software Software reliability metrics Flight control software Software fault tolerance Production of reliable software	UNCLASSIFIED - UNLIMITED SUBJECT CATEGORY 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
UNCLASSIFIED	UNCLASSIFIED	26	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161 NASA-Langley, 1982

National Aeronautics and
Space Administration

Washington, D.C.
20546

Official Business
Penalty for Private Use, $300

SPECIAL FOURTH CLASS MAIL
BOOK

Postage and Fees Paid
National Aeronautics and
Space Administration
NASA-451

[illegible]

NASA

POSTMASTER: If Undeliverable (Section 158 Postal Manual) Do Not Return