Annual Report

Grant No. NAG-1-511

A SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Dr. D. E. Eckhardt, Jr.
FCSD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528235/CS86/102

March 1986

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Annual Report

Grant No. NAG-1-511

A SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Dr. D. E. Eckhardt, Jr.
FCSD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528235/CS86/102

March 1986

Copy No. _____

1. Introduction

Crucial digital systems can fail because of faults in either software or hardware. A great deal of research in hardware design has yielded computer architectures of potentially very high reliability such as SIFT [WEN 78] and FTMP [HOP 78]. In addition, distributed systems (incorporating fail-stop processors [SCH 83a]) can provide graceful degradation and safe operation even when individual computers fail or are physically damaged.

The state of the art in software development is not as advanced. Current production methods do not yield software with the required reliability for crucial systems, and advanced methods of formal verification [GRI 81] and synthesis [PAR 83] are not able to deal with software of the required size and complexity.

Fault tolerance [RAN 75] has been proposed as a technique to allow software to cope with its own faults in a manner reminiscent of the techniques employed in hardware fault tolerance. It is expected that this will provide external performance which will have the required reliability.

The absence of a formal theoretical basis for developing fault-tolerant software has lead to an empirical approach. First generation experiments [KEL 83] [SCO 83a] have provided a proof-of-concept and have shown the feasibility of several fault-tolerant software policies, but these experiments have not yet demonstrated conclusive reliability increases under controlled experimental conditions. Even if reliability improvement had been demonstrated there is no data available showing the size of the improvement

nor any data showing that the resulting reliability is sufficient for crucial applications.

The purpose of the work performed under this grant is to begin to obtain information about the efficacy of fault-tolerant software by conducting a large-scale controlled experiment. The work performed under the current grant reporting period is the planning of the experiment, the preparation of the subject programs, and the definition of a testing procedure for the programs produced during the experiment.

The experiment is being conducted jointly by NASA, four universities, and the Research Triangle Institute. The participating universities are North Carolina State University, the University of California at Los Angles, the University of Illinois at Urbana-Champaign and the University of Virginia. There were several motivations for the use of multiple universities in the experiment. First, it was expected that the diversity in programmer background thus obtained would help avoid correlated errors in the modules produced. Second, the experiment required more qualified programmers than can be recruited from a single institution. Additional benefits arose from the fact that the participants, individually and through previous cooperative endeavors represented most of the previous fault tolerant software experimentation that has been performed in the United States.

By the use of a suitably large set of components, produced in an environment which was carefully controlled to maximize the reliability of each component, we hope to achieve results which are both statistically

significant and relevant to applications such as avionics requiring extremely high reliability. Our goal is to determine the effects of fault-tolerant software on reliability, while controlling or eliminating the effects due to other factors.

In section 2 of this report we review the technical background for the experiment, and in section 3 we describe the refined experiment that has resulted from planning discussions. Section 4 reviews the specific activities at the University of Virginia. Appendix 1 contains the application form used at the University of Virginia in the hiring of students, and Appendix 2 contains the software development protocols proposed during the grant reporting period for the experiment. This document was prepared in cooperation with Dr John Kelly of the University of California at Los Angeles and was a document for discussion only. It is not intended for general dissemination and was the basis for the documented protocol supplied to the various universities by RTI.

Appendix 3 is a proposal for the evaluation of the programs developed in this experiment. It contains input from the various sources identified on its face page, but the development of the document was coordinated and the document was compiled at the University of Virginia.

2. Background

Two different approaches to the tolerance of software faults can be distinguished. They are the *simplex* and the *multiplex* approaches. In the simplex approach various checking procedures are incorporated into the software and provide run-time detection of certain faults [CRI 82]. In the multiplex approach two or more non-identical software modules (versions) are provided that perform the same task.

The sequential implementation of the multiplex scheme is exemplified by the Recovery Block (RB) method [RAN 75], [AND 81]. An acceptance test is performed on the results of the first version; the next version is executed only if the test fails.

The concurrent implementation of the multiplex scheme is employed by the N-version programming method [AVC 77], [CHA 78]. All versions of the software are executed concurrently, and a decision algorithm is applied to the results to determine a consensus. A hybrid scheme incorporating both was proposed by Scott et al. [SCO 83b]

The need for design fault-tolerance in software led to the initiation of a research effort at UCLA in 1975 [AVI 75]. Its goal was to study the feasibility of adapting to software design fault-tolerance the technique of N-fold Modular Redundancy (NMR) with majority voting that was effective in the tolerance of physical faults. The approach was called "N-Version Programming" (NVP) and the first experimental study of its feasibility was completed in 1978 [CHA 78]. A second approach, already under

investigation at the University of Newcastle in England in 1973, was the Recovery Block (RB) technique, in which alternate software routines are organized in a manner similar to the dynamic redundancy (standby sparing) technique in hardware [RAN 75]. The prime objective is to perform run-time software design fault detection by an acceptance test and to implement recovery by taking an alternate path of execution. This technique is also being continuously investigated at several locations. Some comparisons of RB with NVP have been made in [CHA 78], [GAA 80]. A reliability model was proposed by Scott [SCO 84b] and a validation of these models was reported in [SCO 84a]. Several related research activities have been reported more recently, among them [VOG 76], [KIR 76], [LOR 77], [GMV 79], [AND 83], [CRI 82], [CAR 83] are especially relevant.

An experiment to test the fundamental assumption of independence of versions in an N-version system has been conducted jointly by the University of Virginia and the University of California at Irvine. Multi-version software as an approach to fault-tolerant software relies upon independently produced versions failing independently where specification faults are not the cause. The experiment attempts to determine the validity of this assumption using a rigorous statistical approach. No attempt at quantitative assessment of reliability improvement was included.

Twenty-seven versions of a program were prepared by graduate students at the two institutions from a common specification. Extensive efforts were made to ensure that individual students did not cooperate or exchange information about their program designs during the development phase. The

problem definition, preliminary specification, and the standard (i.e. assumed correct) version of the program have all been obtained from the Research Triangle Institute.

3. Refined Experiment

Our primary original goal for this experiment was to determine whether the application of fault tolerance to software increases its reliability *if the cost of production is the same as for an equivalent non-fault-tolerant version derived from the same requirements specification.* The italicized phrase is important and was the key to the significance of the original experiment.

The problem is that if costs of production are ignored, any piece of software can probably be made arbitrarily reliable. Equivalently, if costs are deliberately forced to be very low, any piece of software can certainly be made arbitrarily unreliable. The reason that reliability can always be increased is that exhaustive testing can verify a program and, given enough resources, one can exhaustively test many programs. Similarly, given no cost control, one could hire armies of mathematicians to verify that a program complies with its requirements specification. There are methods for applying verification to simple real-time systems, and if one works hard enough, even floating point arithmetic can be verified. Although these are pathological cases, they illustrate the point that cost is an important factor, and, that unless costs are matched, nothing meaningful can be said about measured *comparative* reliability. A lot could be said about *absolute* reliability but that was not the concern of the experiment. We want to know how to build a system given a fixed budget to achieve the best reliability, and we want to know whether we should employ fault tolerance under those circumstances or not.

During the planning meetings for the experiment, there was some discussion about the possibility of software developed according to a particular methodology reaching some asymptotic reliability level and being unable to surpass it. This is *very* speculative. There does not seem to be any evidence to suggest that this effect occurs. Even if it does, it would be very difficult to prove. The reason the participants in this experiment discussed it is the possibility that some technique, say fault tolerance for example, might allow development of software with reliability above the asymptotic reliability cutoff for non-fault-tolerant methods. If this were the case, there would be a real purpose in evaluating fault tolerance with no attention to cost, but there does not seem to such evidence.

Thus the original goal, and the null hypothesis to be tested in a statistical test of significance was:

> Given a fixed development cost, a fault tolerant software system is more reliable than a non-fault-tolerant software system built from the same requirements specification.

An additional original goal was to produce components that could be combined in various ways to produce different fault-tolerant software configurations, such as N-version or Recovery Block systems, whose performance could be determined by extensive testing.

Even with these ambitious goals, this would have been a preliminary experiment in that many potential variables (such as programming language used) would have been held fixed to remove variability that could influence the reliability of resulting systems.

The problem with these goals is the limited scale of the experiment. To test the various hypotheses, it would be necessary to have many more samples of both fault-tolerant and non-fault-tolerant versions than can be achieved. Various development scenarios were considered in an effort to see how many versions could be produced within the budget. There was some discussion of the possibility of overlapping the two development scenarios by producing the N-version systems first and then testing the individual versions at greater expense to produce a set of non-fault-tolerant versions. Given the hypothesis stated above however, it was agreed that this experiment must be performed in such a way that the significance test is totally fair. Consequently, it was agreed that it is essential that fault-tolerant and non-fault-tolerant development be absolutely independent.

In discussions with the technical monitor, it was agreed that a less ambitious experiment would be useful and feasible with the funds available. The goal of the experiment was modified, therefore, to produce as many versions of a single program as possible and to perform experiments on these programs to test the independence of their faults and the reliability that might be achieved if they are combined into N-version systems. There will be no attempt to produce components to allow comparison of the N-version and Recovery Block strategies, and there will be no attempt to compare fault-tolerant and non-fault-tolerant versions.

4. Activities At UVa

During the grant reporting period, the activities at the University of Virginia have been oriented to preparation for the experiment and the construction of the subject programs. Numerous planning meetings have been attended at the University of Illinois, the Research Triangle Institute, NASA Langley Research Center, and NASA Headquarters in Washington D.C.

Various documents have been prepared for discussion by the participants in the experiment. The most important documents are the development protocol which is reproduced in this report as Appendix 2, and the proposed test plan that is included in this report as appendix 3.

The need to hire graduate students with the highest abilities dictated that the hiring process begin as early as possible. Consequently, an application form was developed at the University of Virginia and the availability of positions for the summer of 1985 advertised. This process was successful and commitments were received from the various students who eventually participated in the experiment at the University of Virginia.

During the summer of 1985, ten graduate students were hired as research assistants and organized into five groups of two. Following the detailed protocol supplied by RTI, these groups prepared programs according to the requirements specification that was also supplied by RTI. The five programs produced were subjected to an acceptance procedure and finally delivered to the coordinator of the experiment at RTI.

REFERENCES

[AND 81]

T. Anderson, and P.A. Lee, *Fault Tolerance, Principles and Practice.* Englewood Cliffs, NJ: Prentice-Hall Intl., 1981.

[AND 83]

T. Anderson and J.C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, 355-364.

[AVC 77]

A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution," *Proceedings of COMPSAC 77*, (First IEEE-CS International Computer Software and Application Conference), Nov. 1977, 149-155.

[AVI 67]

A. Avizienis, "Design of Fault-Tolerant Computers," *AFIPS Conference Proc.*, 1967;-31:733-743.

[AVI 75]

A. Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proc. 1975 Int. Conf. Reliable Software*, 458-464.

[CAR 83]

W.C. Carter, "Architectural Considerations for Detecting Run-Time Errors in Programs," *Proceedings of the 13th Annual International Symposium on Fault-Tolerant Computing*, Milano, Italy, June 1983, 249-256.

[CHA 78]

L. Chen and A. Avizienis, "N-Version programming: a fault-tolerance approach to reliability of software operation", *Digest FTCS-8*, Toulouse, France, June 1978, 3-9.

[CRI 82]

F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, Vol. C-31, No. 6, June 1982, 531-539.

[GAA 80]

A. Granarov, J. Arlat, and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies", *Digest of the 1980 International Symposium on Fault-Tolerant Computing*, Kyoto, Japan, October 1-3, 1980, 251-253.

[GMV 79]

L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment", *IFAC Workshop SAFECOMP 1979*, Stuttgart, May 16-18, 1979.

[GRI 81]

D. Gries, *"The Science Of Programming"*, Springer Verlag, 1981.

[HOP 78]

A.L. Hopkins, et al., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.

[KEL 83]

J.P.J. Kelly and A. Avizienis, "A Specification-Oriented Multi-Version Software Experiment" *Proceedings of the 13th Annual International Symposium on Fault-Tolerant Computing*, Milano, Italy, June 1983, 120-126.

[KIR 76]

K.H. Kim and C.V. Ramamoorthy, "Failure-tolerant Parallel Programming and Its Supporting System Architecture", *AFIPS Conf. Proc.*, Vol. 45, NCC 1976, 413-423.

[LOR 77]

A.B. Long, C.V. Ramamoorthy, et al., "A Methodology for Development and Validation of Critical Software for Nuclear Power Plants," *Proc. COMPSAC 77* (IEEE-CS Int. Computer Software & Applications Conf.), 620-626.

[PAR 83]

H. Partsch and R. Steinbruggen, "Program Transformation Systems", *ACM*

Computing Surveys, Vol. 15, No. 3, September 1983.

[RAN 75]

B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, June 1975, 220-232.

[SCH 83a]

R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", *ACM Transactions On Computer Systems*, Vol. 1, pp.222-238, August 1983.

[SCO 83a]

R.K. Scott, "Data Domain Modeling of Fault-Tolerant Software Reliability", *Ph.D Thesis*, Dept. of Electrical and Computer Engineering, North Carolina State University, 1983.

[SCO 83b]

R.K. Scott, J.W. Gault and D.F. McAllister, "The Consensus Recovery Block", *Proceedings of the Total System Reliability Symposium*, 1983.

[SCO 84a]

R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs "Experimental Verification of Three Fault-Tolerant Software Reliability Models", *Digest of Papers FTCS 14: Fourteenth International Conference On Fault-Tolerant* Computing, Orlando, FL, June 1984.

[SCO 84b]

R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs "Investigating Version Dependence in Fault-Tolerant Software", *AGARD '84*, pp. 21-1, 21-10.

[VOG 76]

U. Voges, "Aspects of Design, Test and Validation of the Software for a Computerized Reactor Protection System", *Proc. of the 2nd Intern. Conference on Software Engineering*, San Francisco, 1976, 606-610.

[WEN 78]

J.H. Wensley, et al., "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.

Appendix 1

Programmer Application Form

NASA Software Reliability Project

Programmer Application Form

(1) Name

(2) Please list undergraduate degree topic(s), GPA, and school awarding the degree(s):

(3) Please list the graduate courses you have taken (with grade) or will take in the spring semester:

(4) With what programming languages do you feel you are fluent? For each, give an assessment of your skill as you perceive it using a numerical scale where '1' represents novice and '5' represents expert. For each, give the length in lines of the longest program you have *successfully* written in that language.

(5) Have you ever been employed as a programmer? If so please summarize your experience:

(6) Please write a brief statement (200 words or less) giving *your* views on why software is not as reliable as we would like it to be (continue on another sheet if you need to).

Appendix 2

Proposed Development Process

THE SOFTWARE DEVELOPMENT PROCESS

AND ASSOCIATED PROTOCOLS

FOR THE REDUNDANT SOFTWARE EXPERIMENT

John Knight and John Kelly

March, 1985.

1. General

This is a working paper on the topics of the title. We will try to present reasonably complete and detailed proposals although, of necessity, they may change and are intended to act as the focal point for discussion at the next meeting of the research group. By *development methodology* we mean the software development methodology employed by the programmers during the software development process. By *development protocol* we mean the mechanics of getting the software developed; the tools used, and how we ensure the things we want get done on time.

In a sense, the development process does not matter a great deal. Whatever results are achieved by this experiment, they will be *conditional* on the development process. Thus any development process would, in principle, be satisfactory. However, if the results are to be believed and regarded as useful by industry, we should adopt a development approach that resembles as closely as possible the methods used by industry. In this experiment, our potential number of versions is already very low and so we had better ensure that every version we pay for is acceptable for analysis.

Protocol on the other hand is *crucial.* If the development protocol fails in some way, for example we cannot guarantee that we have

preserved independence during development or versions are not completed on time, the entire experiment will have been *fiasted.*

The development process is influenced by the students' backgrounds. Can we require that they have all had specific course work? Can we assume they all understand major topics such as abstract data types or structured design? Probably not, and even if we could, there would be other technologies that we would like to use but which are insufficiently known. Differing educational backgrounds is an awkward problem. The solution discussed informally at various meetings is threefold:

(1) Provide each student with a copy of a standard text (Fairley's has been suggested) and require that they read it at the beginning of the experiment.

(2) Run an intense one or two day training seminar at the beginning of the project.

(3) Stop worrying about the problem and assume diverse ability contributes to design diversity.

We are spared the requirements analysis and the preparation of the requirements specification stages of software development since the programmers will be supplied with requirements specification

documents. Also, we assume there will be no post-delivery enhancement or fault correction so there will be no need to consider the phase euphemistically known as "maintenance". Thus, we suggest that development needs to include design, code development, and validation only.

We assume programmers will be working in groups of two in the development phase. We, as a group, have not resolved the issues relating to the development of the voters or assembly of the NMR systems. This is part of the analysis but in practice voters are needed. Perhaps they ought to be developed by the programmers even if we choose to throw them away. For the purposes of discussion, we propose the methodology outlined in the next section and the protocol outlined in the section three.

2. Software Development Methodology

2.1. Background And Development Logging

We need to know who our programmers are. They should fill in a questionnaire detailing their backgrounds. We need to know exactly what is being done when. We propose, therefore, that we require a log be maintained in which each work period is documented. In addition, any logging that can be done automatically should be done. We need to work on that with a Unix wizard.

2.2. Specifications

The experimenters will provide a complete high-level external specification. This will be written in PDL to provide a structured English-like notation. All input and output will be defined through a set of parameters that the program version will use.

At all stages, questions about the specifications will be submitted by electronic mail, reviewed by the experimenters, and responded to by electronic mail (see protocol below). The determination that a question reveals a flaw in the specifications will cause a change to be broadcast to all programmers at all sites. All questions and all responses will be logged.

2.3. Design

We propose using ad hoc design using information hiding and abstract types only. The design will be documented in a form yet to be specified and be delivered on a specified date. A design walkthrough will be required involving only the development team and a report to be produced of the results of the walkthrough. This, and in fact all other walkthroughs, will be attended by the experimenter and/or an aide but with silent participation.

The first deliverable item will be a design document. The content will be a diagram showing the abstract data types and abstraction layers that the team intends to use, a listing including the major data types and variables that the program will use, expressed in Pascal VAR and TYPE parts, the headers of all the procedures that the program will use including the specification on all the parameters, and a comment explaining the procedures purpose. This document will be due on a date yet to be specified.

2.4. Code Development

Code development will be done in Pascal using coding standards provided by the experimenters. The code will be developed up to system compilation only, i.e. there will be no "random" executions of the entire program. Unit testing will be performed on the individual parts as they are written. Code walkthrough will be required involving only the development team and a silent observer, and a report will be produced of the results of the walkthrough.

The program will be developed in a strict top-down fashion in which each layer of the abstraction will be implemented and tested as a unit using stubs for the incomplete lower layers. The second deliverable will be a series of compiled programs representing the results of the top down development at each abstraction layer. Testing of each layer will be by a small number of ad hoc tests that

the team deems suitable. The team will be responsible for developing the necessary test drivers. These tests will be aimed at removing the major flaws in the layer only.

2.5. Validation

Validation will be performed by testing only, and will be limited to functional testing.

A test plan and test log will be required. The test plan to be documented and delivered on a specified date. The test log to be documented and delivered on a specified date. Test drivers to be developed for each of the three test phases by the team to assist in the test process, but again these are to be the only software tools used in validation.

Once the entire source text is prepared and integrated, the program will be validated according the test plan. All test executions during validation must be logged. The completed log is the fourth deliverable item. The final program is the fifth deliverable item.

2.6. Acceptance Testing

Acceptance testing is our determination of whether the software is of adequate quality to be used in the experiment. The specification of the form of the acceptance test is not part of the development

process. The action to be taken following failure is. Naturally, we require that the delivered software satisfy the acceptance test at the end of the development process. In the event of failure, we propose that the programmer be required to document his actions in his development log *in detail*; every design change, every changed line of code, every recompilation, every re-executed test. We also require that the programmer keep trying until they have passed the acceptance test no matter how long it takes.

3. Development Protocol

There are several aspects to be considered in the development protocol. We am not even sure what all of them are let alone how they should be resolved. However, here are our suggestions:

(1) Prior to the experiment, all programmers will be given three presentations during which there will be *no* questions (seems a little extreme). Questions will be posed and answered by electronic mail. The presentations will be on the application, the goals of the experiment and the associated protocols, and the software tools and facilities they are to use.

(2) All code development to be on VAX's running UNIX.

(3) All code to be written in Pascal using PC with the -S option. Somehow we will have to enforce coding standards. The standards will be needed more to ensure portability than code quality.

(4) All documents to be prepared using TROFF.

(5) All communication between the programmers and the experimenters during the experiment to be by electronic mail and all communication be logged by the experimenter.

(6) All due dates and all necessary documentation will be provided at the start of the experiment.

(7) Logs showing the activity of the group members will be turned in weekly.

(8) Working hours for the programmers will be flexible but at least forty hours per week of effort is required.

4. Issues To Resolve

Here is a list of issues in the development process and protocol areas that I feel we need to discuss at the next meeting. Of course, everybody is encouraged to add to this list as they see fit.

(1) What procedures are we going to follow and what rules are we going to enforce to maintain development independence?

(2) In what form should the documentation we require be presented? If we determine that there are flaws in a particular part of the development (for example, a design is inadequate) should we do anything to correct the situation. In a practical environment, the programmers would be faced with management and customer reviews as they went along. Do we want to try to model this?

(3) What questions do we put in the background questionnaire?

(4) What form should the development log take? How do we ensure its kept accurately? Do we really care or need it (of course we do)?

(5) What detailed restrictions on language elements should be imposed? This is most important if we are going to ensure portability to many machines for testing.

(6) Should any other software tools should be used, required, permitted? If so, which other tools?

(7) What approach should be used in synchronizing events to ensure all the teams work at roughly the same rate and that deliverables are available on time?

Appendix 3

Proposed Tesing Plan

THE SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE

Part Two

Analysis Of The Multiple Version Software

Alper Caglayan, CRA
Roy Campbell, UIUC
Dave Eckhardt, NASA-LaRC
John Kelly, UCLA
John Knight, UVA
Dave McAllister, NCSU
John McHugh, RTI
John Pierce, RTI

Compiled By:
John C. Knight

Department of Computer Science
University of Virginia
Charlottesville
Virginia, 22903

DRAFT FOR DISCUSSION ONLY

1. INTRODUCTION

This research group has generated twenty programs in part one of this project. Part two, which is described here, is an analysis phase in which the programs will be studied and research results obtained.

The programs seem to be in more-or-less reasonable shape. The preliminary results obtained by John Kelly indicate that the programs might not be good enough for the analyses that is proposed. This second part may need to incorporate a "maintenance" phase in which the various programs are passed through a second, more elaborate, acceptance test.

This document outlines the process of the second part of the experiment. The document is organized as follows. In section 2, the procedures that will be followed are outlined, and in section 3 the goals of the experiment are described. Section 4 considers the problem of determining when a test has been passed, and section 5 discusses types of testing. Section 6 describes the testing environments that are required, and section 7 addresses test case selection. Data collection is considered in section 8, the mechanics of testing in section 9, data storage and distribution in section 10, performing the tests in section 11, and analyzing the results in section 12. Finally, a plan of action in the form of a proposed sequence of events is presented in section 13.

2. PROCEDURES

The testing approach defined in this document is to be viewed as the *phase one* analysis of the programs. It is designed to reveal the faults that we suspect the programs contain, to give quantitative reliability information, and to achieve the basic goals of the experiment. All the institutions that participated in the preparation of the programs are interested in the results of this phase and will receive the raw data as it is collected if they wish. The results of this phase will be deposited in AIRLAB for the benefit of the research group as a whole. Phase one is to be a cooperative effort in which the analyses performed will be jointly defined and jointly undertaken. Any publications resulting from these analyses will be joint.

When phase one is complete, the original goals of the experiment will have been achieved and some useful assessment data for redundant software will have been produced. More elaborate and diverse analyses that are not part of the *original* goals can and should be done on these programs. This more elaborate analysis will be viewed as *phase two* analysis. Different aspects of phase two might be undertaken by individual researchers as they see fit, and phase two is *not* the subject of this document.

For the foreseeable future, the programs and the raw phase-one results will only be available to members of the research group. This restriction will protect the research interests of the group.

3. GOALS OF THE EXPERIMENT

The overall goal of this experiment is to assess the performance of the multi-version approach to fault-tolerant software. Our concern is with software for critical avionics (and similar) applications and so performance here means reliability primarily, although there are other factors of interest.

The goals of the experiment dictate the testing that has to be done. There are four primary objectives for the testing. They are:

(1) To obtain empirical estimates of the reliability of the programs *individually* and in various *combinations*. This data is important in order to be able to make quantitative estimates of the effects of multi-version systems on overall reliability.

(2) To determine what faults the programs contain. Clearly the characteristics of the faults themselves are important. The mistakes made by the programmers need to be identified and categorized to allow determination of how they might have been prevented or located during testing. Tests designed to locate the faults as quickly and easily as possible need to be performed.

(3) To determine the performance of the programs from the perspective of the controls' engineer. There are difficulties that might arise in these programs that the computer scientist might regard as serious (i.e. a failure) but that the controls' engineer would ignore because it

is a pathological case. And vice versa. Tests need to be run that allow the controls' engineer to make a determination of the adequacy of the programs' performance.

(4) In performing this testing, there is an opportunity to obtain data that is not obviously required, i.e. *ancillary* data. It would be foolish not to collect this data since its inexpensive to get and might be needed, or at least it might allow interesting related experiments or analyses.

A fundamental issue in performing any testing program is the evaluation of the results of individual tests on individual programs. The traditional issue in testing of finding an "oracle" arises. The oracle problem for these programs is discussed below.

The various goals of this experiment are not equivalent in the tests that they require. For example, locating the faults is a lot different from determining the reliability. The testing necessitated by each of the experiment's goals is also discussed below.

4. TESTING ORACLE

A definition of *failure* (or alternatively *success*) on any given test case is required. For the different tests that address the various goals of the experiment, the definition of failure will differ slightly, as discussed below. However, in general, an *individual* program will be deemed to have failed if one of the following occurs:

(a) The program experiences an execution-time failure such as an attempt to take a negative square root.

(b) The program enters an infinite loop.

(c) Any of the outputs produced by the program differs from the "correct" value by more than a predetermined amount (as yet unspecified). The outputs that will be checked are those listed in the specification document as required outputs. Intermediate calculations will not be checked, although they may be recorded.

(d) The program identifies incorrectly that sensors have failed.

Given this definition, it is necessary to be able to categorize the results of a test according to the definition. This is quite difficult for the RSDIMU programs individually, and extremely difficult for multi-version systems built from the RSDIMU programs.

For the individual programs, two different approaches will be used depending on the circumstances of the particular test. First, the RTI reverse algorithm will be used to determine the sensor values that would have generated a randomly-generated acceleration estimate. This approach will be modified by adding random noise, etc, as necessary.

The second approach is to use the FORTRAN version of the program prepared by Charles River Analytics (modified as necessary) as a *gold* version of the program. For the purposes of preliminary analysis, it can

be assumed that if a subject program disagrees with the gold program, the subject program is wrong. In the long run, it is not necessary to assume that the gold version is perfect, although it is likely to be of very high quality. Provided the twenty programs are of reasonable quality, the cause of discrepancies with the gold program can be determined when they occur.

It will not be possible to operate without a gold program using the majority value of the program versions as a presumed correct result. The different outputs generated by groups of programs that John Kelly observed in his preliminary testing would make it very difficult in general to decide which of multiple values was to be taken as correct. Indeed, more than one value may be acceptable in many cases and it would be a mistake to consider that any program had failed in those circumstances.

For multi-version systems built from the individual programs, a great deal of information is required for each test case. For each test case, it is important to know which of the following occurs:

(1) All of the versions produce acceptable outputs.

(2) A sufficient number of the versions produce acceptable outputs that the system output is acceptable. In this case one or more faults are being tolerated.

(3) An insufficient number of the versions produce acceptable results and no output can be produced by the system. In this case the error is

being detection but faults are not being tolerated. This is still a useful property of a multi-version system, in fact it is the only goal of a 2-version system.

(4) A sufficient number of the versions produce apparently acceptable outputs that the selection procedure produces a system output but this output is, in fact, wrong and so unacceptable. In this case the faults are not being tolerated and the error is not being detected. This is the worst possible situation.

To determine which of the above has occurred for any testcase being executed by a multi-version system, either a complete vote has to be performed after all versions have been executed or all the outputs have to be saved so that voting can be simulated at a later time.

5. TYPES OF TESTING

5.1. Reliability Assessment

Reliability assessment requires data on the performance of the programs in an operational setting. It might be the case that a program with a known fault never fails because that fault is never manifested. Correspondingly, a fault considered relatively obscure might be manifested for each cycle on a long input sequence because every element of the sequence causes the fault to manifest itself. These are the factors that

affect our perception of reliability.

Testing to measure reliability needs to be performed in an environment that simulates the real operational environment as closely as possible. This is the only way to get believable reliability data. Charles River Analytics has a simulator of a Boeing 737 available and it is proposed that this, combined with a flight scenario simulator, be used to subject the programs to simulated flight conditions.

Each flight would consist of takeoff, cruise, and landing phases. The cruise phase, which is usually the longest, would be shortened deliberately since if a program works correctly during part of the cruise phase it is likely to work correctly over all of the cruise phase.

For the purposes of reliability estimation, a test case will be defined to be a complete simulated flight. A program will be defined to have failed for that test case if, at *any* point during a simulated flight, it meets the definition of failure given in section 4.

5.2. Fault Location

Fault detection is merely the process of determining that a program contains a fault. The theory of testing is not sufficiently mature that it dictates the process to be followed. The literature contains a lot of papers on testing and there is quite a folklore surrounding testing. However, many of the papers are theoretical and propose approaches that

cannot be used. For example, *domain testing* [1] is a nice idea but is only defined for programs that do not use arrays and have only linear predicates. Loops complicate things too. Other more formal methods have been proposed by several people in the research group but most of these other methods require a very sophisticated tool to be built. For example, branch testing requires the monitoring of branches in the program. The group does not have access to tools to perform this monitoring on all of the equipment that might be used for testing.

A method is required that can be applied with the available tools, that is applicable to the subject programs, and that requires almost no human intervention. The first requirement eliminates most of the more sophisticated methods and the last requirement eliminates even functional testing, stress testing, and the like.

Random testing could be used exclusively to locate the faults. This approach has the advantage that it does not require much effort in building the environment nor in operating it. There is also the marginal advantage that the software to do this kind of thing exists already at RTI and UCLA. The disadvantage is that it takes a lot of computer time and it might not locate the faults very easily, perhaps not at all. However, well-structured random testing seems to be the best choice at this point. A recent paper [2] suggests that random testing might be quite satisfactory.

There are certain dimensions of the input space that are sufficiently narrow that exhaustive testing can be used in those dimensions. For example, there are not a large number of sensor failure combinations and so all possible combinations can be tested quite easily. For the purposes of discussion, these inputs will be referred to as *limited-input* quantities. More sophisticated testing could be part of the phase two analysis.

For the purposes of fault location, a test case will be defined to be a single execution of a program, i.e. calibration followed by processing of a single set of data. A program will be defined to have failed for that test case if, during that execution, it meets the definition of failure given in section 4.

5.3. Control Performance Assessment

This is different from reliability assessment and fault location. What is required is data on the performance of the programs in an operational environment but where the data of interest is the functional performance of the programs. It might be the case that a program appears to be operating correctly when viewed by a computer scientist but the controls' engineer may perceive some weakness in its operation. This weakness is most likely to be in some numeric aspect of the problem.

5.4. Ancillary Data

The ancillary data items that seems to be needed, and the reasons for needing them are:

(1) The floating point outputs of the programs. These are needed for all the other goals but they can be used to study the distribution of the numbers, to do experiments with various forms of voters, to analyze numerical accuracy (or lack thereof), etc.

(2) The execution time for *each* test for *each* program. One of the concerns with N-version programming is the possibility of holding up fast versions while slow versions complete. This has to be detected and distinguished from versions that have died entirely. Capturing the execution times of these programs would give unique empirical data on the problem of diverse execution times.

(3) The statements executed in each program on each test. There is little-to-no data on how much of a program is used during production executions. This would be very valuable information to have and it can be gathered (albeit with some difficulty) as the tests are run for the other major goals.

6. TESTING ENVIRONMENTS

Two separate test environments are required. The first will be designed to perform testing according to some of the established practices in the literature, particularly random testing, and it will be designed to do fault location. This environment will operate by selecting random quantities from those input space dimensions that are large and, as far as possible, testing all possible combinations of the limited-input quantities. The design of this environment will be such that very little *reliability* performance data on the programs will be produced, but extensive empirical measurements of *failure probability* will be obtained. The environment will operate in a largely unattended manner provided there are no catastrophic failures of the control software. It will operate on many computers in parallel in order to reduce the elapsed time required to obtain large numbers of test cases.

The second environment will be a simulated production environment. It will be designed to perform tests that simulate the inputs that the programs might receive when operating on a commercial air transport. The environment will also foster stress testing where stress is derived from the application domain, such as very high noise, various combinations of sensor failures, extreme values of vehicle acceleration or movement, etc. Long duration tests of the programs under the operational conditions provided by this environment will provide reliability data.

The second environment will also be used by the controls' engineers to get the performance data that they need. They could subject the programs to any scenario that interests them and get information on the corresponding controls' performance of the programs.

Clearly there is some overlap in the needs of these two environments. There will be quite a lot of shared software, and each will produce extensive amounts of output that will have to be captured on tape.

7. TEST CASE SELECTION

The actual test cases to be used are determined by the major goals of the experiment. The fourth goal (ancillary data) does not require any test cases of its own since all the ancillary data will be collected as part of the main testing process. Thus only the first two goals need be considered in selecting test cases.

7.1. Reliability Assessment

One characteristic of these programs is that they perform calibration and then process a single set of sensor values. This is not typical but was done to remove the history element from the definition of a test case. In retrospect, this was a mistake in that, as discussed above, to do any meaningful reliability assessment, the programs must be subjected to

a realistic operational environment.

For reliability assessment, realistic operation will have to be simulated by either modifying the programs (probably not permitted) to allow one calibration to be followed by multiple samples, or supply the programs with identical calibration data for each sample. The latter assumes there are no faults that will manifest themselves by subtle interactions among these many redundant calibrations.

Once the calibration issue gets resolved, large numbers of simulated "flights" will be executed with these programs doing the acceleration estimates. Each flight scenario will be repeated with each of the limited-input variables varied over its entire range, and with a selection of different calibration data sets.

7.2. Fault Location

For fault location, all the non-limited-input quantities will be varied over their entire range, and the limited-input quantities will take on all possible values. Thus, for example, the accelerometer outputs, the number of operational sensors, the noise characteristics, and so on will be varied over their entire ranges. This approach will generate *unrealistic* conditions in that the inputs might not describe events that could take place in an aircraft. However, since the purpose is fault location, it is reasonable to expect the programs to perform as required in the specifications. Any failures produced by processing unrealistic inputs will

reveal faults that may or may not be important to realistic operation. They need to be found in any case.

The test scenario will then be that each limited-input quantity will be set to an appropriate value and then some large number of tests will be run with that setting. Non-limited-input quantities will be generated at random from within the defined range and with an agreed-upon distribution of values. This process will then be repeated for each combination of the limited-input quantities.

8. DATA COLLECTION

The data items produced will be similar for both environments, although for the simulated production environment a single calibration and a long series of sensor values will be processed for each test case. The potential data output for a simulated flight test is therefore huge. Also, for the simulated flight tests, it will be necessary to associate the physical states of interest with each test case. This will have to be done separately.

Clearly for the input, only the random number seed used to generate the input values for the non-limited-input quantities and the values of the limited-input quantities need be recorded. Except for the few limited-input quantities, the actual data can always be reconstructed from the seed if needs be. The outputs that are collected will be limited by available storage space. Storing all the outputs, execution times, etc

would be preferred but this is not practical. However, some complete sets of output data will be collected to allow a reduced form of the analysis that needs the entire output, such as timing analysis. For a single random test, the complete output dataset would consist of:

(1) The values of all the outputs defined in the specifications; namely LINOFFSET, LINNOISE, SYSSTATUS, LINFAILOUT, LINOUT, BESTEST, CHANEST, CHANFACE, DISMODE, DISUPPER, and DISLOWER.

(2) The statement execution counts.

(3) The execution time for the testcase.

Recording of both the complete input and output values requires several hundred bytes per program per test case. However, note that since all the input quantities are generated by the driver, they can be reconstructed from the seed and needn't be stored. For all twenty programs, the amount of data to be stored if we assume that inputs can be reconstructed from the seed will be approximately 5,356 bytes per test case. The computation of this estimate is shown in figure 1.

If we add execution traces, we need to add 27,200 bytes (one 8-bit integer for each executable line in the programs). This adds 27,200 bytes to each case and reduces the number that may be stored by a factor of about six. It is, therefore, probably unrealistic to keep execution traces for any but a few representative cases.

OUTPUT VARIABLES

VARIABLE NAME	DESCRIPTION	# BITS	
LINNOFFSET	array [1..8] of real	512	
LINNOISE	array [1..8] of boolean	8	
SYSTATUS	boolean	1	**
LINFAILOU	array [1..8] of boolean	8	
LINOUT	array [1..8] of real	512	
BESTEST	RECORD consisting of:		
	status - enumerated type	2	**
	acceleration - array [1..3] of real	192	
CHANEST	array [1..4] of BESTEST-type records	778	
CHANFACE	array [1..4] of 0..6	12	**
DISMODE	16-bit integer	16	
DISUPPER	array [1..3] of 16-bit integer	48	
DISLOWER	array [1..3] of 16-bit integer	48	
TOTAL		2,135	

This corresponds to 267 8-bit bytes.
For 20 programs this is a total of 5,340 bytes for output.

** Total bytes are computed assuming that SYSTATUS, BESTEST.status, AND the last 4 bits of CHANEST are encoded into seven bits and stored in one byte.

INPUT VALUES	
Random number seed, real	64
EXECUTION STATS	
Execution time, real	64
	128

Total data per test case is - 5,356 bytes.

Figure 1 - Data Storage Required

We can get approximately 100 MB per tape at 6250 bpi, so we can store 100,000,000 / 5,356 cases per tape or 18,670. Ten tapes is a

reasonable number to purchase and store so about 186,700 tests seems like a feasible number to record completely and make available for analysis.

More than 100,000 tests will have to be run. A goal of several million is reasonable. For the test cases in which all of the outputs are not recorded, the stored outputs will be limited to success or failure, and the type of failure. Clearly, in this case absence of outputs can be used to indicate success in the recorded results so only the failure cases need be recorded. The cause of the failure (execution-time failure, disagreement with an output, etc) can be coded very compactly. Similarly, the values of the limited-input variables do not need to be recorded for every test case, merely when they change. Thus for each test, all that needs to be recorded is the random number seed used together with the result from each program in coded form, and these quantities only need to be recorded when one of the programs fails for any particular case. If at least one program fails on 10% of the tests, and the results of a test can be coded in one byte, the results of about 50,000,000 tests can be recorded on a single tape reel (very rough estimate).

It is probably unrealistic to record anything except success or failure for the simulated flight tests since they will produce so much output.

In all of the above, it is assumed that multi-version systems will be tested as well as the single program versions. As outlined in section 4,

this means that the various categories of results for a multi-version system will have to be recorded, the voting procedure will have to be defined and implemented ahead of time, and all the votes actually performed for each test. The only alternative is to execute the programs individually, record all the outputs, and simulated the voting at a later time.

9. THE MECHANICS OF TESTING

First of all, testing must be reproducible, and must be reproducible on *different* machines. It is essential that a test case that is deemed to have caused a failure at one site be reproducible at another site. This is necessary to ensure that raw testing can be performed wherever computer time is available, but analysis can be performed wherever the researcher who is interested has his analysis tools. Similarly, testing must be capable of parallel operation on *different* machines. This is necessary to ensure that all available computers can be used in parallel no matter what site they are located at.

Since different machines use different floating point formats and algorithms, reproducibility is unlikely to be achieved exactly but using the floating point hardware available will come close. A much bigger problem is the variability in random-number generators. Clearly different installations will use vastly different generators and so it is essential that the testing be done by test harnesses that include their own random-number generators and that these generators be tested on the anticipated

equipment before any testing begins.

One of the quantities to be recorded in the test cases where all the outputs of interest are recorded is execution time. This is machine dependent, and, since relatively few test cases will be providing complete sets of output, these tests can all be run on the same machine.

Dedix has the facilities to allow the programs to be executed in parallel and be subjected to various voting scenarios at the points where votes are to be taken. Assessment of the performance of various voting algorithms and the behavior of the programs with different voting algorithms is not a primary goal of this experiment, but, as noted above, we need to be able to vote as the tests are performed. Dedix is a desirable facility but it is also important to be able to execute tests at various sites that do not have Dedix and that do not have equipment that could run Dedix. Thus, the test harnesses will have to include specialized voting software that is specific to this project; software that Dedix provides but that is required at multiple sites.

10. DATA STORAGE AND DISTRIBUTION

As tests are performed, the site performing the tests will report the status of the testing to the other interested members of the group on a weekly basis, and transmit the results of the tests on tape to AIRLAB as they are generated. Tapes will be duplicated at AIRLAB as they become available and transmitted to the interested members of the group.

AIRLAB will function as a repository for the results of all the tests and as coordinator of the testing activities.

11. PERFORMING THE TESTS

Equipment for performing the tests is available at UCLA, UVA, and CRA. Each of the two forms of tests will be performed at whichever site has computer facilities available. As an initial allocation of responsibility, the random testing will be performed at UCLA, and the flight simulations will be done at UVA and CRA. All sites will try to get the necessary support software (test harnesses, etc) running.

12. ANALYSIS OF TEST RESULTS

The raw data resulting from the testing process is not itself very useful. The results need to be processed to obtain the required results.

As outlined in section 3 the first goal is to determine the quantitative improvement in reliability that would be obtained by using multi-version software. This can be done by simulating the effect of running combinations of programs in parallel.

13. SEQUENCE OF EVENTS

The sequence of events that need to be undertaken as part 2 of this experiment are:

(1) Determination of the suitability of the subject programs for analysis. If the programs are as bad as is indicated by the preliminary UCLA tests, then a recertification and maintenance phase will need to be undertaken.

(2) Determination of the limited-input variables and their set of possible values.

(3) Construction of a random-number generator that is portable. Testing of this generator on all the computers that might be used in the test process.

(4) Determining how many random numbers will be needed for each test case, running the random-number generator through all the random numbers that will be needed for *all* the tests to check for cycles, and recording of the random number that occurs at the beginning of the sequence that will be used for each set of 5,000 tests. This latter information will allow any sequence of 5,000 tests to be run on any machine at any time and for all results to be coordinated. This process needs to be performed for both the random tests and the simulated flight tests.

(5) Construction of a test harness from the original RTI test harness for performing random testing. This test harness will need to be a considerable extension of the RTI software given the limited variation that RTI performed on many parameters as reported by UCLA.

(6) Installation of the CRA 737 simulator and the CRA version of the program (to be the gold program) on all computers that might be used for simulated flight testing.

(7) Determination of the set of flight profiles that will be used in the simulated flight tests.

(8) Implementation of data management software that will allow the large volumes of data to be catalogued, stored reliably, and simply distributed to all the interested parties.

(9) Preparation of software to analyze the raw results to produce meaningful data on reliability improvement.

(10) Evaluation of the testing process by executing the two testing environments on limited numbers of tests at the various sites.

(11) Evaluation of the data distribution and storage mechanisms by using them to process the results of the testing evaluations.

(12) Execution of many millions of tests.

(13) Analysis of the results of the tests.

In all of the above, it will be continually necessary to modify the approach and so regular meetings and teleconferences will be required. In addition, regular communication by electronic mail so that all parties are involved in part 2 of the experiment is essential.

14. CONCLUSION

Two separate testing environments need to be built to perform the tests to obtain data for each of the four goals. The specific test that are run in each environment need to be determined in more detail described here.

This is a working document. Please forward all comments and changes to John Knight (jck@uvacs) for inclusion in the next version.

15. REFERENCES

(1) L.J. White, and E.I. Cohen, "A Domain Strategy For Computer Program Testing", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3.

(2) J.W. Duran, and S.C. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665 Attention: Dr. D. E. Eckhardt, Jr. FCSD M/S 130
4 - 5*	NASA Scientific and Technical Information Facility P.O. Box 8757 Baltimore/Washington International Airport Baltimore, MD 21240
6 - 7	J. C. Knight
8	A. Catlin
9 - 10	E. H. Pancake/Clark Hall
11	SEAS Publications Files

*One reproducible copy

JO# 7261:lsl

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 560 There are 150 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science Within these disciplines there are well equipped laboratories for conducting highly specialized research All departments offer the doctorate, Biomedical and Materials Science grant only graduate degrees In addition, courses in the humanities are offered within the School

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 16,400), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.