DEVELOPMENT OF A SOFTWARE SAFETY PROCESS

AND

A CASE STUDY OF ITS USE   *NAG1-1123*

IN-61-CR

Annual Progress Report   *168975*

P-26

Submitted to:

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

Attention:

Dr. D. E. Eckhardt, M/S 478

Submitted by:

John C. Knight
Professor

SEAS Proposal No. UVA/528344/CS93/103
June 1993

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF

ENGINEERING
& APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

DEVELOPMENT OF A SOFTWARE SAFETY PROCESS

AND

A CASE STUDY OF ITS USE

Annual Progress Report

Submitted by:

John C. Knight
Professor

Department of Computer Science
School of Engineering and Applied Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903-2442

TABLE OF CONTENTS

1. INTRODUCTION

The goal of this research is to continue the development of a comprehensive approach to software safety and to evaluate the approach with a case study. The case study is a major part of the project, and it involves the analysis of a specific safety-critical system from the medical equipment domain. The particular application being used was selected because of the availability of a suitable candidate system. We consider the results to be generally applicable and in no way particularly limited by the domain.

With more and more important functions in existing and proposed safety-critical systems being implemented by computers, concern over the role of software in such systems has increased. An especially important area is that class of systems for which safety rather than reliability or availability is the overriding issue. Some research that addresses the safety of software specifically has been reported but many open questions remain. In particular, no complete process is available for engineers to follow when building applications software for systems in which safety considerations dominate. We are developing such a process through a combination of theoretical and empirical research.

The research is concentrating on issues raised by the specification and verification phases of the software lifecycle since they are central to our previously-developed rigorous definitions of software safety. The theoretical research is based on our framework of definitions for software safety. In the area of specification, the main topics being investigated are (a) the development of techniques for building system fault trees that correctly incorporate software issues and (b) the development of rigorous techniques for the preparation of software safety specifications. The results of research to date is documented in a latter section of this report.

A second area of theoretical investigation is the development of verification methods tailored to the characteristics of safety requirements. Verification of the correct implementation of the safety specification is central to the goal of establishing safe software. Our experience to date has shown that, for certain classes of safety problems, exhaustive testing of a system is possible in reasonable amounts of time. Similarly, we have shown that a complete test set for certain properties can be derived automatically from the safety specifications if these specifications are written in a suitably formal notation such as 'Z' [11].

The empirical component of this research is focusing on a case study in order to provide detailed characterizations of the issues as they appear in practice, and to provide a testbed for the evaluation of various existing and new theoretical results, tools and techniques. The system being used in the case study is the *Magnetic Stereotaxis System* (MSS), a safety-critical medical system presently under development. The overall, long term approach being taken in the empirical research using this system is to develop fully functional software of sufficient quality to be suitable for safety-critical use. This approach is necessary to ensure that the research undertaken is not weakened by unrealistic assumptions or restrictions. The empirical research is implementing the various techniques resulting from the theoretical research and using these implementations to assess the the theoretical results.

The remainder of this report is organized as follows. In the next section, the Magnetic Stereotaxis System is summarized. Previous related work is summarized in section 3. The research results to date are reviewed in section 4. Presentations given on the general topic covered by this grant and listed in section 5, and publications resulting from this grant are summarized in section 6.

2. THE MAGNETIC STEREOTAXIS SYSTEM

The Magnetic Stereotaxis System (MSS) is an investigational device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Washington [15]. It operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field. By varying the magnitude and gradient of the external field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor. The device can be used for hyperthermia by radio-frequency heating of the seed from an external source or for chemotherapy by using the seed to deliver drugs to a site within the brain. The MSS concept promises to be far less traumatic to the patient than present invasive approaches to such treatments. However, the concept is new and its success depends on satisfactory completion of several magnetic systems engineering tasks (now well underway).

The externally applied magnetic field of the MSS has to have a gradient of at least five Tesla/meter in order to move the seed through brain tissue. This field is produced by a set of six superconducting coils that are mounted in a cryostatic enclosure that surrounds the patient's head during surgery. Even using superconducting coils, achieving the field required within the small available space is one of many research issues that have to be resolved.

The seed location within the brain is monitored in real time by an X-ray imaging system that provides two perpendicular images that include views of the skull, a set of fiducial markers, and the seed. Data from these images is combined with a set of stored, pre-operative Magnetic Resonance Images (MRI's) and displayed for the neurosurgeon. The seed is maneuvered to the region of interest within the brain by varying the

externally applied magnetic field.

The MSS includes a computer system that is used:

- to control the X-ray, RF-heating, and electromagnetic subsystems,

- to present MR images and X-ray data to the neurosurgeon, and

- to accept the neurosurgeon's input and translate it into commands to the various subsystems.

Clearly, the MSS is a safety-critical system. The greatest concern is, of course, with patient injury. A patient could be injured by failure of any of the physical subsystems. For example, failure of the seed-positioning coils could move the seed incorrectly and result in serious brain damage. Similarly, failure of the X-ray or RF-heating subsystems could injure the patient by excessive radiation exposure or tissue heating.

Cryogenic System

Coil Control System

X-Ray Imaging System

Operator Display

Computer Control System

X-Ray Sources

Superconducting Coils

Radio Frequency Heating Coils
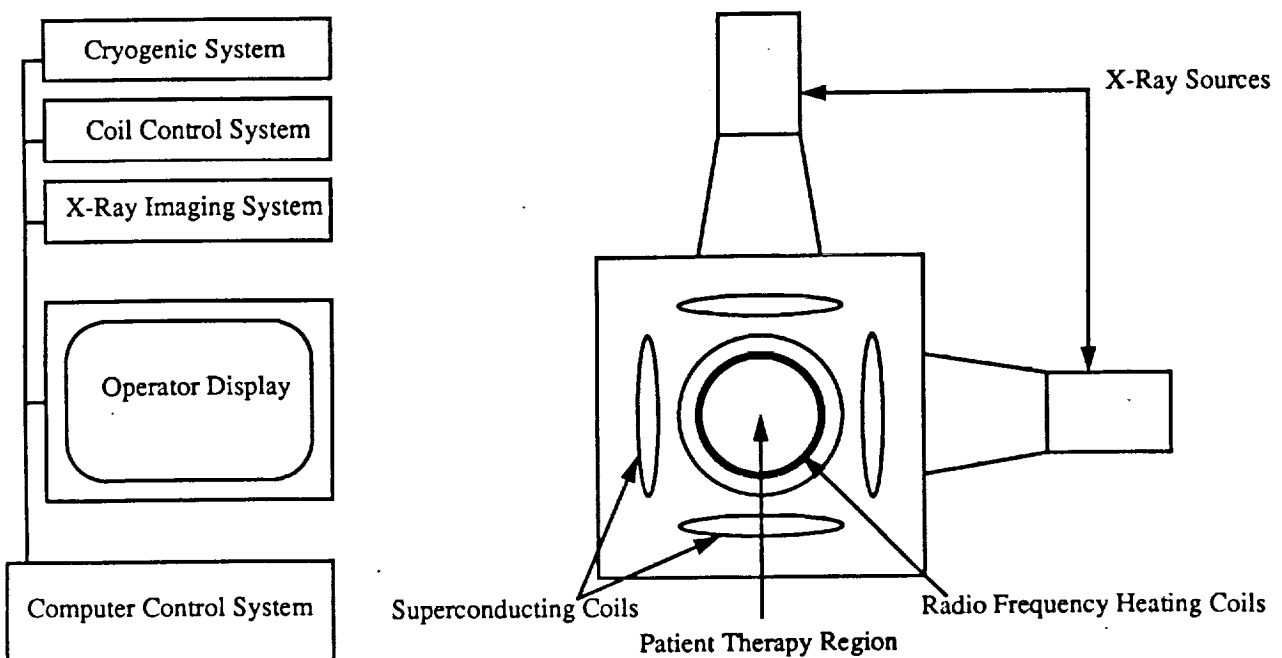
Patient Therapy Region

Fig. 1 - Magnetic Stereotaxis System

Just as subsystem failure could lead to injury, so could failure of the control software. For example, a software failure could cause incorrect drive currents to be applied to the seed-positioning coils thereby resulting in brain damage. Even a failure of the image display system could lead to injury if the surgeon's actions were based on improperly displayed MR images or X-ray data.

Although patient injury is the greatest concern, protection of the equipment itself from damage resulting from incorrect use or internal failure is also important. For example, the seed-positioning coils are very high energy devices and the current through them has to be managed very carefully. Inappropriate current changes could quench the coils and cause damage to the ancillary cryogenic equipment.

The state of the MSS is that the concept is fully defined, the majority of the basic research in physics is complete, and a fully functional prototype is nearing completion for demonstration and evaluation. It will be operated by prototype software [15]. We using the MSS as a case study in our research on safety-critical systems. Our goal is the development of tools and techniques that permit such systems to be built with the required assurance of safe operation. Figure 1 show the major components of the MSS.

3. PREVIOUS WORK

We have previously defined a preliminary framework of definitions for software safety. This framework was deemed to be a necessary first step in the development of the comprehensive software safety process that we seek. In addition, we have previously identified certain elements of the software-safety process that we are developing. This previous work is summarized here briefly as background for the section on research results.

3.1 Definitional Framework

The definitional framework views a system as a set of interacting components [4]. As has been noted elsewhere [4, 5], software is never unsafe in isolation. But, in practice, software is never used in isolation. Software is always used within a system, and it is merely one of many components of a complete system. It is a *component* of a complete system in the same sense that entities such as computer hardware, sensors, actuators, power supplies, packaging, and even human operators are each merely components of a system.

In a general sense, none of these various components is unsafe in isolation because, when isolated, these entities are separated from the notion of hazard. It is only in the context of the complete system that the various hazards have meaning. None of the components changes and suddenly becomes "unsafe" when incorporated into a system. Any design deficiency in a component is present when the component is isolated just as when it is part of a system. A deficiency in a component leads to a hazard only when the component is part of a system because it is the system that defines the hazard. Hazard is a system concept, not a component concept. In particular, as noted by Leveson the notion of hazard is not a software concept [5].

Since software is a system component, an important goal in the development of a theory of software safety is separation of the theory from systems safety yet smooth integration with it. Without separation, we run the risk of needlessly complicating and weakening our conclusions about software with external systems concerns. Without integration, we run the risk of reaching conclusions about software that do not contribute to the safety of the associated systems.

The basic structure of the definitional framework requires three sets of specifications for each system component. The first defines the overall functionality of the component, the second states the assumptions that can be made about the component's behavior should it fail, and the third specifies the action that the component is to take when other components fail. Any component in a safety-critical system is expected to comply with all three of these sets of specifications. From the perspective of component failure, software is no different from any other component.

Within this framework, the systems engineer has the formal role of preparing a set of specifications for the software, and the resulting software is defined to be safe if it complies with these specifications. Once it is developed, showing that software is safe according to this definition is, therefore, an exercise in *verification*. The software is safe, in this formal sense, to the extent that this verification is successful.

In a strictly formal sense, the software engineer's role is to implement the software and show that the safety specifications developed by the systems engineer are implemented correctly. The formal role of the software engineer is limited in this way because *this is all the software engineer is qualified to do*. Informally, however, the software engineer is encouraged to contribute to the development of the specifications, to analyze the specifications, to report anything that appears to be a deficiency in the specifications, and generally be on the lookout for any defect in the system that could detract from safe operation. These are not *formal* roles of the software engineer, however, because he or she is unlikely to be qualified to accept them.

3.2 Software-Safety Process

Our research on the software-safety process is in two complementary areas. The first is concerned with developing a rigorous approach to determining and specifying the software safety requirements. The second is focusing on means of assessing the software safety using both formal verification techniques and empirical testing methods. The various theoretical approaches that are being developed are inspired by the case study. Similarly, the effectiveness of these approaches are being assessed using the case study.

Previous approaches to software safety have failed to stress the importance of a separate specification of the software safety requirements. Instead, the implementation of the software has sometimes been made to double as the safety specification. Although this approach eliminates the need for a separate safety specification, it makes the process of verification unnecessarily difficult. Ultimately, it is only the verification of software safety that is important. It is the specification that makes such a verification possible

A separate safety specification provides a basis for precise communication between the software engineer and the systems engineer. It also permits the exploitation of the precise definition of software safety given in the definitional framework we are developing. Recall that, within this framework, the systems engineer has the formal role of preparing a set of specifications for the software. The resulting software is defined to be safe if it complies with these specifications, and, therefore, formally showing that software is safe is an instance of verification. The software is safe, in this formal sense, to the extent that this verification is successful. It should be noted that verification here might involve many technologies including testing and inspection. It is because of this definition that the our research focuses on the specification of safety and the subsequent

verification of the correct implementation of those specifications.

At present, system fault tree development for systems involving software is an ad hoc process of refinement in which few, if any, provisions are made in the process for the software. The process under development aims to develop system fault trees that incorporate software systematically as system components.

It is clear that the essence of the software safety specification is contained in the system fault tree. Despite this, there is no rigorous method for deriving the software safety specification from the fault tree nor verifying that a software safety specification accurately models the intentions of the associated nodes and structure of the system fault tree. We are investigating the possibility of deriving the system safety specification from the system fault tree and using the derivation to show that the software safety specification is complete and unambiguous.

Another key aspect of the process is verification. We have demonstrated the concept of deriving a set of test cases from the formal safety specifications. In principle, if such a test set could be derived and shown to be complete for any given system, the correct execution of the test set would constitute a proof that the safety specification was being met by the software implementation.

Although it has been argued that the correctness of software cannot be established for safety-critical systems by testing [1], this is a general result to which there are exceptions. We are of the opinion that safety is just such an exception for two reasons. First, safety specifications are typically smaller than functional specifications, and second, time can be accelerated during testing of safety facilities because they are called upon to act so infrequently during normal operation. Thus testing is being developed as an important aspect of the safety process under development.

4. RESEARCH RESULTS TO DATE

The research results achieved during the grant reporting period are summarized in this section. Documents mentioned are listed in detail in section 6.

4.1 Reliability Assessment

Previous work on software reliability assessment by life testing in which error detection is imperfect has been been revised. The results are documented in a paper that will appear in the *IEEE Transactions on Software Engineering*.

4.2 Definitional Framework

The definitional framework of software safety that is the basis of the software-safety process under development has been extended and refined. The revised framework is documented in a technical report.

4.3 Phased Inspections

An extensive experimental evaluation of phased software inspections and the associated toolset has been completed and documented. The research will be published in a special issue of the *Communications of the ACM* on software quality that will appear later this year.

4.4 Experience With Formal Specification

Our preliminary experience using the formal specification notation Z for the specification of the software for the MSS has been documented. The paper was presented at the 7th Annual Z User Group Meeting and appears in the proceedings.

4.5 Reuse Of Specifications

As part of our research on formal specifications, we have developed a prototype library of reusable specification parts. This library and some observations about the issues in the reuse of specifications were reported in a paper at the Fifth Annual Workshop on Institutionalizing Software Reuse and appears in the proceedings.

4.6 Developing Test Cases From Specifications

Extended results in the development of test cases directly from formal specifications in Z have been obtained as part of a file-system case study. These results are documented in a technical report published by George Mason University.

4.7 Prototype Software For The Magnetic Stereotaxis System

In order to have a better vehicle for experimentation, a completely new prototype software system is being developed for the Magnetic Stereotaxis System. The new prototype is a distributed implementation operating on as many computers as are required for the computational load. The new software architecture permits new functionality to be added easily as required. In addition, the new prototype is implemented in C++ and used the standard X-windows graphic interface and Motif widget set.

4.8 Software Safety Process

A major goal of this research program is to develop a process by which software specifications can be developed in a rigorous and complete manner. The present state of the development of this process is that the overall structure has been drafted but most of the effort during the grant reporting period has been in developing support techniques

for the process.

Three key support techniques are being developed. They are *Information Flow Analysis, Software Failure Emulation*, and *Tri-State Models*. Each of these techniques is summarized here.

Information Flow Analysis

A difficult problem in software specification for safety-critical systems is determining all of the different system failures for which the software might have to take action. For example, complex peripheral equipment such as sensors and actuators can fail in a variety of different ways and the software's response has to be carefully specified in every possible case.

In some situations, the software specification is determined easily. For example, the complete failure of a sensor would require the software to respond by employing a backup sensor or different control algorithm. However, in practice there is no systematic way to determine whether all possible failure scenarios to which the software must respond have been identified. As an example of the difficulty, consider the X-ray imaging system from the MSS. The following failures to which the software must respond have been identified:

- An X-ray source could fail on.

- An X-ray source could fail off.

- A fluoroscope on which an image is captured could fail and return no image.

- A fluoroscope could fail and return an outdated image.

- A fluoroscope could fail and return a defective image containing phantom objects that are actually failed pixels in the fluoroscope.

- The communication system between the control computer and the X-ray system could fail and supply incorrect commands or defective images.

All of the above are failure modes with which the applications engineer should be aware. However, many different technical areas are involved and no systematic technique exists presently to derive complete and consistent software specifications from such complex physical systems. We have developed a solution to this problem called *Information Flow Analysis*.

One of the key concepts in *system* safety analysis is flow analysis. Hazards are determined by considering potential undesired energy flows. Component networks are used to model other systems flows such as coolants, hydraulic fluids, and electrical power. However, *information flow* is an altogether different notion that has yet to be fully explored. While the disruption of information flow has been considered, the corruption of information flow has been largely ignored. And while the disruption of information flow is important in that it permits the software to continue monitoring and controlling the physical devices, it is largely a systems engineering concern. Redundant cables and the elimination of single points of failure generally suffice to ensure that information flow will continue uninterrupted.

But such analyses have minor, if any, impact on the software safety specification. It is much more important from the perspective of software specifications, to ensure that all possible points of information corruption have been properly identified. If a device failure can corrupt critical system data, the possibility exists that this faulty data could propagate throughout the system and eventually be consumed by the software.

Information Flow Analysis is a technique in which all sources of information are determined for a system, and, for each source, the entire path from the information source to the computer is identified. The path is then examined and, for each component that modifies or transmits the information, all possible failure modes are identified. The effect of each failure mode on the information is determined and, to the extent possible, software specifications are prepared to detect the occurrence of the failure and to counter its effects.

Using the MSS example once again, two important pieces of information for the system are the X-ray images. The sources of this information are the X-ray emitters themselves. The information starts out as X-ray photons that pass through the patient's head and then through a lengthy set of processing hardware units before the images are finally present in the memory of the control computer. The flow analysis of this particular information identifies all possible failure modes of all possible devices that might affect the X-ray images and determines an appropriate set of software specifications in each case. Dealing with phantom objects on the images, for example, is essential since a phantom could appear to be a seed or marker and lead to incorrect image location. The software specification in this case to perform extensive reasonableness checks on the returned image.

In general, there are two general classifications of information corruption: those that can be detected and those that cannot. Information Flow Analysis determines all the possible sources of information corruption and partitions them according to their detectability. For those failures that can be detected, additional software (and perhaps hardware) requirements will be generated. Those failures that cannot be detected will either involve a failed device or faulty software. Those that involve failed devices will require either higher reliability or redundancy in these devices. Those that involve

failed software will require that techniques be applied to reduce the probability of software failure in that case to an acceptable level.

Software Failure Emulation

The primary thrust of Information Flow Analysis is to determine the various component defects outside of the software for which the software might have to take action. *Software Failure Emulation* is a complementary technique that defines the elements of the software that are critical to safe operation. The combined output of Information Flow Analysis and Software Failure Emulation is the complete software safety specification.

Software Failure Emulation is a two-phase technique that determines in a rigorous fashion the elements of the safety requirements that are dictated by software. The technique takes advantage of the fact that systems engineers impose a certain level of formality on software safety when they perform systems safety analyses such as fault-tree analysis. A system fault-tree analysis[*] is intended to uncover all of the combinations of system occurrences that could lead to a system hazard. The technique originated from the need to identify the likely hardware degradation faults and combinations thereof that could lead to a hazard. As such, a system fault tree documents in a systematic way the various events of which the systems engineer needs to be aware. Unfortunately, typical system fault tree theory and practice do not include software.

The key concerns with software in a safety-critical system are:

---

[*] It is extremely important to distinguish between *system* fault trees and *software* fault trees. The former are tools used by systems engineers to determine system hazards. The latter is an informal software verification technique.

- that it will not respond correctly when required, or

- that it will initiate actions when not required to that lead to a hazard.

The latter concern arises because any software-initiated activity in some other component of a system must be considered as a failure mode for that component. For example, if the software could switch on an X-ray device at an inappropriate time, it is reasonable to assume that the device might also have switched itself on. If the software fails to turn the device off, this appears no different from the device refusing a request to turn off. In this sense, the software makes the device appear to have failed when in fact it has not. Analysis based on this observation derives from the fact that software can give the appearance of, i.e., *emulate*, component failure, hence the name of the technique.

The first phase of Software Failure Emulation is the determination of which system activities the software can initiate and which it can detect. The second phase of Software Failure Emulation consists of determining all of these possible failure modes methodically. This is done as part of an extended form of fault tree analysis.

There is one subtle but significant difference between this fault tree analysis and conventional fault tree analysis that must be considered here. This difference is in addition to the fact that the actual fault trees must be retained for further analysis, rather than being discarded after the probability of failure is computed. This difference is the fact that a software failure can actually effect many devices concurrently and that the probability of failures must be ignored when building the fault trees. In conventional fault tree analysis, construction of subtrees is halted when it becomes clear that the probability of a given branch is low enough to be ignored. These extremely low probabilities of failure are usually caused by coincident, independent failure of several very reliable devices. Since the already low probabilities of failures are multiplied to

determine the probability of the combined event, these values can be vanishingly small and safely ignored. However, as has been noted above, software can emulate the failure of the device and no probability of failure has yet been assigned to that software. In addition, the presence of controlling software can eliminate the assumption of independence and so system fault trees involving software-controlled devices must be completed beyond the point where they would normally be truncated.

Tri-State Models

Tri-state modeling is a technique that is related to Failure Modes and Effects Analysis, Event Tree Analysis, and system Fault Tree Analysis. The technique is still under development but appears at this stage to be a general technique that includes the three other forms of analysis as special cases.

Tri-state Models are used to document and analyze all of the manners in which modules and devices can affect one another, either via direct control or by passing information that could affect the operation of the recipient. Tri-state models depend on Module Interaction Graphs which are hierarchical structures. They begin at the highest level with individual nodes representing the software and the various major system components, including the user if present. Lower levels are expanded to reveal software component objects and physical subsystems respectively. Arcs to and from components can be directed to the entire component, or to one or more sub-components within. These graphs are similar to state-charts, but with the entries representing objects rather than states. A complete module interaction graph represents the complete information flow throughout all the components (including software) of the system.

The arcs coming from a node in a module-interaction graph are the three transitions that a system can take as a result of the effects of the component represented

by the node. The three transitions are (a) correct operation, (b) erroneous operation that was detected, and (c) erroneous operation that was not detected. Since they represent state transitions over time that a system experiences, the module interaction graph is actually a particular form of Markov model. As such, Tri-state models are based on Markov analysis.

Module Interaction Graphs make several simplifying assumptions. The first of these is that all undetected errors are of equal severity. This assumption ensures that we can use a single state to represent the possibility of any combination of failures. Although this might not appear to be true, the context in which this information is later used will ensure that this does not introduce difficulties. The second assumption is that if an error can be detected later in the flow, it can be detected immediately after it is introduced. This assumption will be weakened in subsequent work. The assumption ensures that extra states to handle all possible latent failures will not need to be introduced.

The transitions between the first and third states are fairly straightforward, but the transitions dealing with the detected failures are more involved. There are several ways to deal with a detected failure. If the failure involves a detected operator error or a failure of a device that can be reset or replaced, the option exists to repeat the failed operation. If the failure can be recovered from via alternate functionality, this alternative can be invoked. Another alternative is to report the failure as a failure of the module that is under investigation. That these alternatives correspond exactly to the exception handling concepts of restarting, recovering, and propagating is not a coincidence. In the event that either the detection, recovery, or propagation mechanism fails, the detectable error becomes an undetectable error.

The creation of the module interaction graphs is in itself a useful endeavor. It requires the systems engineer to consider methodically every possible alternative that could affect the operation of a given component. Each possible exceptional condition would have to be individually considered and dealt with. The module interaction graph also provides a convenient mechanism for precisely documenting these considerations. The resulting graphs also represent a formal entity (a Markov model) that can be subjected to formal manipulation. If each transition is assigned a percentage possibility, it is trivial to calculate the percentage chance of an error passing undetected through the system. Furthermore, a fault tree can be derived representing a complete expression of how a device failure could occur. These fault trees could be incorporated into the system fault trees in order to explore common-cause failures.

We are continuing the process of defining and evaluating the techniques that are suggested by the concept of Tr-state Models.

5. PRESENTATIONS GIVEN

During the grant reporting period, in addition to the presentations made at conferences listed in the following section, presentations were given at the following locations:

- Praxis PLC, Bath, England.

- University of Southampton, Southampton, England.

- Naval Research Laboratory, Washington, DC.

- Food and Drug Administration, Washington, DC.

- Medical College of Virginia, Richmond, VA.

- West Virginia University, Morgantown, WV.

- College of William and Mary, Williamsburg, VA.

- University of Houston, Clearlake, TX.

6. RESEARCH PUBLICATIONS

Several publications have either been prepared, published, or accepted for publication during the grant reporting period. In this section, these publications are listed. Copies of each publication have been supplied to the sponsor under separate cover.

(1) P.E. Ammann, S.S. Brilliant, and J.C. Knight, "The Effect of Imperfect Error Detection on Reliability Assessment via Life Testing", *IEEE Transactions on Software Engineering*, to appear.

(2) J.C. Knight and E.A. Myers, "An Improved Software Inspection Technique and an Empirical Evaluation of Its Effectiveness", *Communications of the ACM*, to appear.

(3) D.M. Kienzle and J.C. Knight, "Reuse of Specifications", *Fifth Annual Workshop on Institutionalizing Software Reuse*, Palo Alto, CA, November 1992.

(4) D.M. Kienzle and J.C. Knight, "Preliminary Experience Using Z To Specify A Safety-Critical System", *Seventh Annual Z User Group Meeting*, London, ENGLAND, December 1992 (proceedings published by Springer Verlag).

(5) P.E. Ammann and J. Offutt, "Functional and Test Specification for the MiStix File System", technical report ISEE-TR-93-100, George Mason University, Fairfax, VA, January 1993.

(6) J.C. Knight and D.M. Kienzle, "Safety-Critical Computer Applications: The Role of Software Engineering", Technical Report TR-92-23 (revised), Department of Computer Science, University of Virginia, Charlottesville, VA, January 1993.

REFERENCES

[1] Butler, R.W. and G.B. Finelli, *The Infeasibility of Experimental Quantification of Life-Critical Software Reliability*, SIGSOFT '91 Conference on Software for Critical Systems, New Orleans LA, Dec. 1991.

[2] Green, A., "Safety Systems Reliability", *John Wiley & Sons, New York*, 1983.

[3] Jahanian, F., and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, Sept. 1986.

[4] Leveson, N., "Software Safety: Why, What, and How", *Computing Surveys*, Vol. 18, No. 2, June 1986.

[5] Leveson, N., "Software Safety in Embedded Computer Systems", *Communications of the ACM*, Vol. 34, No. 2, Feb. 1991.

[6] Leveson, N., S. Cha, and T. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees", *IEEE Software*, Vol. 8, No. 4, July 1991.

[7] Leveson, N., and P. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, Sept. 1983.

[8] Leveson, N., and J. Stolzy, "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, March 1987.

[9] Parnas, D., A. van Schouwen, and S. Kwan. "Evaluation Standards for Safety Critical Software", *Technical Report 88-220, Dept. of Computing and Information Science, Queen's University, Ontario*, May 1988.

[10] Sayet, C., and Pilaud, E., "An Experience of a Critical Software Development", *Proc. of 20th Int. Conference on Fault Tolerant Computing*, Newcastle Upon Tyne,

DISTRIBUTION LIST

1 - 3   National Aeronautics and Space Administration
Langley Research Center
Hampton, VA   23665

Attention: Dr. D. E. Eckhardt, M/S 478

4   National Aeronautics and Space Administration
Langley Research Center
Acquisition Division
Hampton, VA   23665

Attention: Mr. R. J. Siebels
Grants Officer, M/S 126

5 - 6*   NASA Scientific and Technical Information Facility
P. O. Box 8757
Baltimore/Washington International Facility
Baltimore, MD   21240

7 - 8   E. H. Pancake, Clark Hall

9 - 10   J. C. Knight

11   J. M. Ortega

**   SEAS Postaward Administration

12   SEAS Preaward Administration Files

*One reproducible copy
**Cover letter

JO#5169:ph