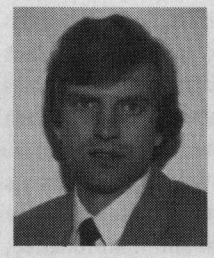
termination of the manpower peak. The ratio of manpower peak to completion date is quite precise and is the best method of determining the delivery data. Cumulative manpower plots are particularly unsuited for any type of predictions, changing most slowly at the time when one desires the most information. We have shown how valuable management information can be obtained by laying out the data in a systematic manner.

The manpower data by itself should not be used to drive the project. The data are used for clues that point to the real status and problems. The key to success in this aspect of modeling is the establishment of reliable and identifiable milestones and an understanding of their relationship to the overall project.

Acknowledgment

W. J. Wesner of Raytheon managed the project described in Fig. 1. His contributions to the analysis of the data are gratefully acknowledged. The author would also like to thank the referee for constructive suggestions.



Roger D. H. Warburton was born in Cardiff, Wales. He received a degree in physics from Sussex University, Sussex, England, and the Ph.D. degree in physics from the University of Pennsylvania, Philadelphia, where he studied under a Thouron Scholarship.

After working at Raytheon, Portsmouth, RI, for several years, he became Manager of Software Technology, specializing in software cost estimation, CAD tools for signal processing, and factory test languages. He is currently Manager of Operations for JAYCOR, Middletown, RI.

Analyzing Software Safety

NANCY G. LEVESON AND PETER R. HARVEY

Abstract—**With the increased use of software controls in critical real-time applications, a new dimension has been introduced into software reliability—the "cost" of errors. The problems of safety have become critical as these applications have increasingly included areas where the consequences of failure are serious and may involve grave dangers to human life and property. This paper defines software safety and describes a technique called software fault tree analysis which can be used to analyze a design as to its safety. The technique has been applied to a program which controls the flight and telemetry for a University of California spacecraft. A critical failure scenario was detected by the technique which had not been revealed during substantial testing of the program. Parts of this analysis are presented as an example of the use of the technique and the results are discussed.**

Index Terms—**Fail-safe software, fault tree, real-time software, safety verification, software reliability, software safety, software validation, system safety.**

Manuscript received July 30, 1982; revised January 20, 1983. This work was supported in part by Contract 7-656146-T-DS with Hughes Aircraft Company and by a joint grant of the University of California MICRO Project and Hughes Aircraft Company. The project is receiving additional support from the System Development Corporation.

The authors are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

Introduction

In recent years, advances in computer technology have gone hand in hand with the introduction of computer usage in new application areas. The problems of safety have become critical as these applications have increasingly included areas where the consequences of failure are serious and may involve grave dangers to human life and property. Computers currently control reactions in nuclear power plants, track airplane positions in air traffic control systems, monitor patients in intensive care units of hospitals, deal with the complexities of space flight in aerospace programs, and control military and defense systems.

There is growing concern about errors in these systems. Much of the focus in software research has been on techniques to eliminate errors prior to the operational use of the software system [4]. For the most part, software errors have been regarded as a temporary problem which will disappear as soon as adequate methodologies for program development and validation can be devised. But progress in developing these methodologies has been slow, and error-free software may not be a

0098-5589/83/0900-0569$01.00 © 1983 IEEE

realistic goal. Almost all these methodologies must be used by humans, and humans are fallible. Even automatic program synthesis may not be the panacea we seek. Errors are made in all stages of program production. There is evidence, in fact, that more errors occur during the requirements and design stages than during coding [14].

Current validation methods present some serious problems in assuring reliable software. The greatest limitation of validation techniques is that they cannot guard against imperfect execution environments, and in fact verification often assumes "perfect" execution environments. Furthermore, a program can only be validated for given environmental assumptions. Software faults may be caused by undetected hardware errors such as transient faults causing mutilation of data, security violations, human mistakes during operation and maintenance, errors in underlying or supporting software systems, or interfacing problems with other systems such as timing errors. Just as all correct paths through a program cannot be tested for a complex system, neither can all environmental conditions.

Since removal of all faults and perfect execution environments cannot at this point in time, and perhaps never will, be guaranteed using these methods, there is incentive to make software fault-tolerant. In this approach, it is assumed that run-time errors will occur, and techniques are used to attempt to ensure that the software will continue to function correctly in spite of the presence of errors. As examples of this approach, see [3], [16], [17]. The software fault-tolerance procedures which have been proposed, however, cannot guarantee 100 percent reliability. In fact, the added complexity of providing fault-tolerance may itself cause run-time failures (e.g., the timing problems caused by backup redundancy procedures on the first Space Shuttle flight [5]).

Thus it appears that run-time failures which are related to software faults remains a problem and is likely to remain so for some time. In situations where the "cost" of a failure is high, special techniques must be employed to make the system "safe." This paper describes a safety analysis technique which can be used to help verify that the logic contained in the software design will not produce system safety failures or, alternatively, to find failure modes or failure scenarios related to software faults which could lead to catastrophes. The results of the analysis can be used to guide further design of the software, to guide in the placement of run-time assertions and software exception-handling and redundancy procedures, to determine the run-time conditions under which fail-soft or fail-safe procedures should be called, and to facilitate thorough testing by pinpointing critical functions and test cases. The technique is adapted from one originally developed to measure hardware safety called fault tree analysis. The successful results of applying software fault tree analysis to a flight and telemetry control program for a University of California spacecraft are described.

Definitions

The first step in dealing with safety is to separate failures into safety and nonsafety failures. Using the definition of a *failure* as the inability of a system to perform a required function within specified limits, a *safety failure* will be defined as a failure which leads to casualties or serious consequences. The definition of "serious consequences" will have to be left up to the system designer. It obviously includes injury or death. It may also include the destruction of property or any undesirable consequence the designer of the system considers to be as or more important than the correct operation of the system.

This last point is important since the handling of safety failures will differ from that of nonsafety failures, and, on occasion, the response to a safety failure might actually reduce the reliability of the system since the response will focus on reduction of risk rather than attainment of mission [11], [12]. For example, a nonsafety failure will probably be handled first by an attempt to recover, and if this is not possible, then by a run-time reconfiguration of the system (including the software) to eliminate the failed or erroneous noncritical component in such a way that critical functions are maintained. Safety failures, on the other hand, usually involve critical functions which cannot be reconfigured out of the system. If recovery is not possible after a safety failure, then fail-soft or fail-safe procedures must be activated. A *fail-soft* system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed. The goal of a *fail-safe* system, in contrast, is not to continue functionality but to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

The goal of software safety is the avoidance of system safety failures which are caused by a software error and/or are detected and handled by software procedures. A *safe system* is one which prevents unsafe states from producing safety failures where an *unsafe state* is defined as a state which may lead to a safety failure unless some specific action is taken to avert it. In order to prevent safety failures, it will be necessary to be able to detect unsafe states. Software fault tree analysis can be used to aid in this process.

Software Fault Tree Analysis

Software fault tree analysis (SFTA) is a technique to analyze the safety of a software design. The technique is an offshoot of an engineering technique used for the safety analysis of electromechanical devices. SFTA proceeds in a manner similar to hardware fault tree analysis and uses a subset of the symbols currently in use in the hardware counterparts. Thus hardware and software trees can be linked together at their interfaces to allow the entire system to be analyzed. This is extremely important since software safety procedures cannot be developed in a vacuum but must be considered as part of the overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, the environmental failure may cause the software error to manifest itself. In many previous safety mishaps, e.g., the nuclear power plant failure at Three Mile Island, the safety failure was actually the result of a sequence of interrelated failures in different parts of the system.

The first step in any safety analysis is a hazard analysis of the entire system. This is essentially a listing and categorization of the hazards posed by the system. The classifications range from "catastrophic," meaning that the hazard poses

extremely serious consequences, down to "negligible" which denotes that the hazard will not seriously affect system performance. Once the hazards have been determined, fault tree analysis proceeds. It should be noted here that in a complex system, it is possible, and perhaps even likely, that not all hazards can be predetermined. This fact does not decrease the necessity of identifying as many hazards as possible, but does imply that additional procedures may be necessary to ensure system safety.

The goal of SFTA is to show that the logic contained in the software design will not produce system safety failures, and to determine environmental conditions which could lead to the software causing a safety failure. The basic procedure is to assume that the software has performed in a manner which has been determined by the hazard analysis will lead to a catastrophe and then to work backward to determine the set of possible causes for the condition to occur.

The root of the fault tree is the event to be analyzed, i.e., the "loss event." Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are unable to be analyzed for some reason.

Building Fault Trees

The symbols used to build software fault trees are shown in Fig. 1. The symbols chosen are a subset of those used for hardware systems in order to allow maximum integration of techniques. Fault tree analysis can be used at various levels and stages of software development. Fig. 2 is an example of a high-level system fault tree including both hardware and software components. It is described in more detail below. Each of the leaf nodes may be analyzed further depending upon how the tree is to be used and the corresponding stage of software design. At the lowest level the code may be analyzed. This paper will focus on the code level analysis, but it should be noted that higher levels of analysis are important and can and will be interspersed with the code level. Thus the analysis can proceed and be viewed at various levels of abstraction. It is also possible to build fault trees from a program design language (PDL) and to thus use the information derived from the trees early in the software life cycle.

When working at the code level, the starting place for the analysis is the code responsible for the output. The analysis then proceeds backward deducing both how the program got to this part of the code and determining the current values of the variables (current state). This type of backward reasoning about programs is, of course, a well-understood and researched subject [1], [9]. Therefore, only a few example control structures will be examined with respect to the fault trees they produce.

Fig. 3(a) shows how the analysis proceeds back through an if-then-else statement. It is assumed that the event has occurred within the bounds of the statement. In other words, executing the statement in some environment caused the event, so we wish to build the tree which describes that environment. Fig. 3(b) shows the fault tree for the simple statement "if $a > b$ then $x := f(x)$ else $x := 10$" when analyzed for the event "$x > 100$."

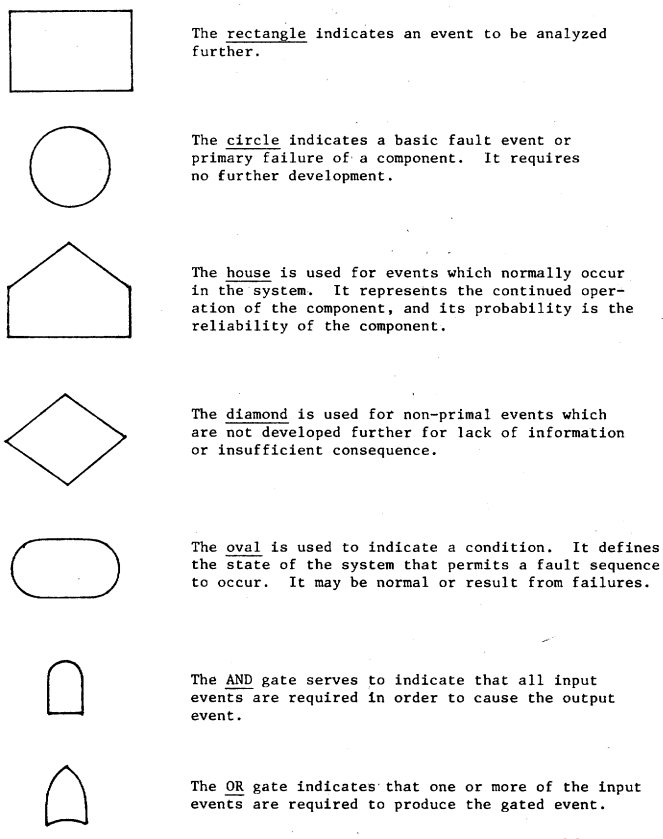


Fig. 1. Relevant fault tree symbols from MIL-STD-882A.

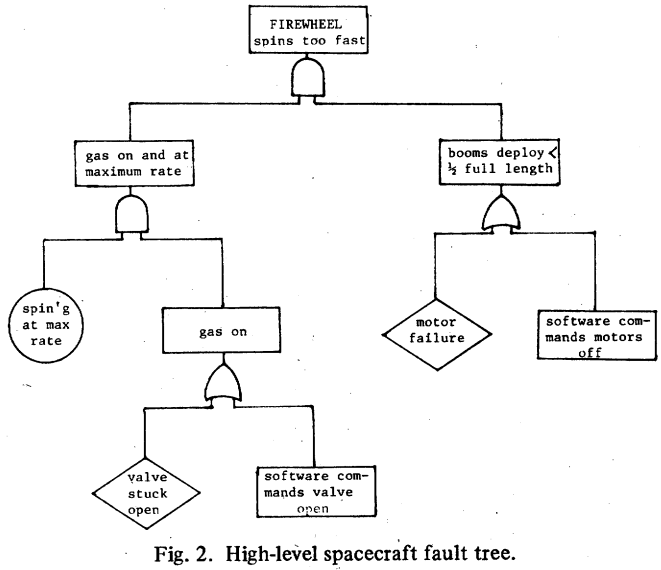


Fig. 2. High-level spacecraft fault tree.

Note that the right subtree describes an impossible situation. This subtree then can be assigned a zero probability and immediately pruned from the tree.

The fault tree of Fig. 3(b) has basically transformed the problem of analyzing the loss event into subproblems. The analysis could stop here and assertions placed in the code to detect the unsafe state, or the software which operates prior to the statement can be analyzed for the events "$a > b$" and "$f(x) > 100$." A complete analysis will generate the full set of faults and conditions necessary for the loss event.

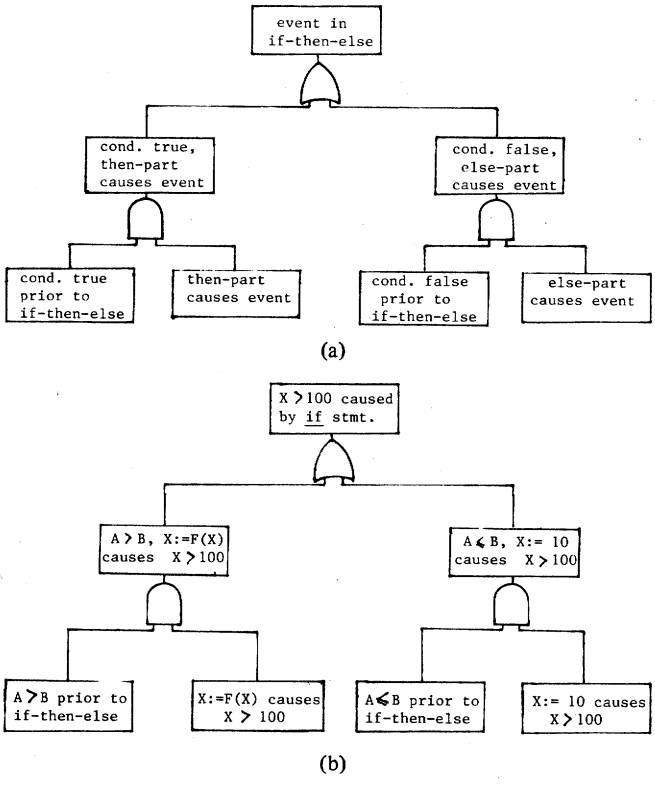


Fig. 3. (a) General fault tree format for if-then-else. (b) Example of if-then-else structure in fault tree.

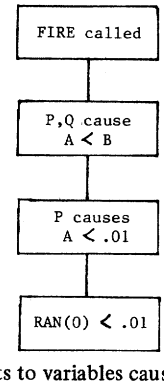


Fig. 4. Assignments to variables cause goal replacement.

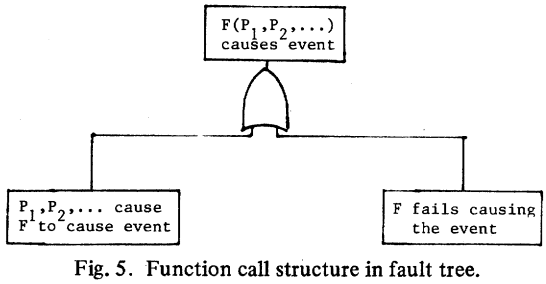


Fig. 5. Function call structure in fault tree.

Fault trees for assignment statements, procedure calls, and while statements are shown in Figs. 4-6. Fig. 4 demonstrates the analysis of the three line program:

p: a: = ran (0);

q: b: = 0.01;

if $a < b$ then fire

for the event "fire called." As shown in Fig. 5, erroneous output from a function call may result from parameter or algorithmic faults.

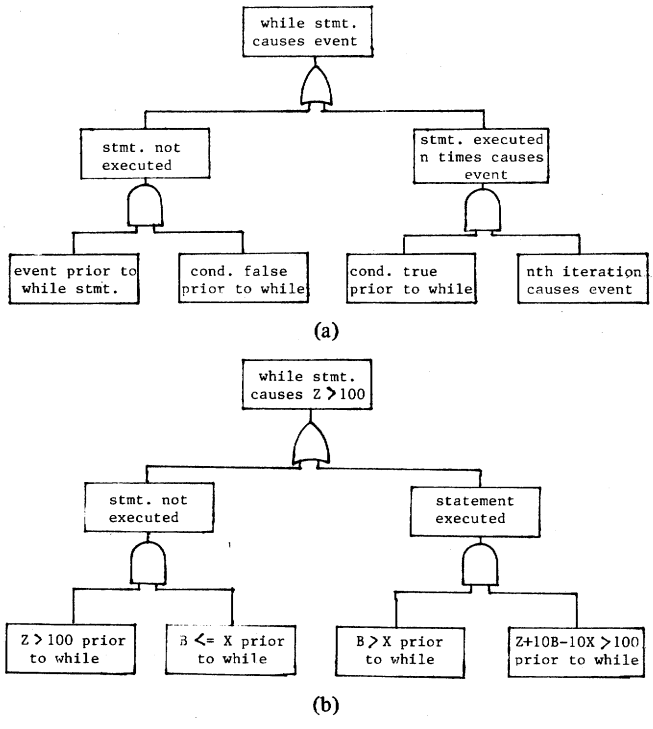


Fig. 6. (a) Format for while statement. (b) Example of while statement.

The format for the fault tree of a while loop is shown in Fig. 6(a). An example is shown in Fig. 6(b), i.e., the analysis of the while statement:

while $b > x$ do

begin b: = b - 1;

z: = z + 10;

end

for the event "$z > 100$." The left subtree of the figure assumes that the while statement never executed. This implies that z had to be greater than 100 initially and b less than or equal to x. The right subtree examines the modification of z within the body of the while statement. Letting n represent the unknown number of times the loop will execute and letting z_n be the value of z on the nth iteration, then $z_n > 100$ is the event to be analyzed within the loop. It is now necessary to find an expression for n and z_n in terms of z_0 (the original value of z before the loop). Examining the second assignment statement of the loop, the loop invariant with respect to z can be seen to be

$$z_n = z_0 + 10 * n \tag{1}$$

and after n iterations of the loop, $b_n = b_0 - n$. But $b_n = x$ since iteration stopped after n times, and hence,

$$b_0 - n = x$$

$$b_0 - x = n. \tag{2}$$

Combining (1) with (2)

$$100 < z_n <= z_0 + 10\,(b_0 - x).$$

One further point should be made. Control is difficult to

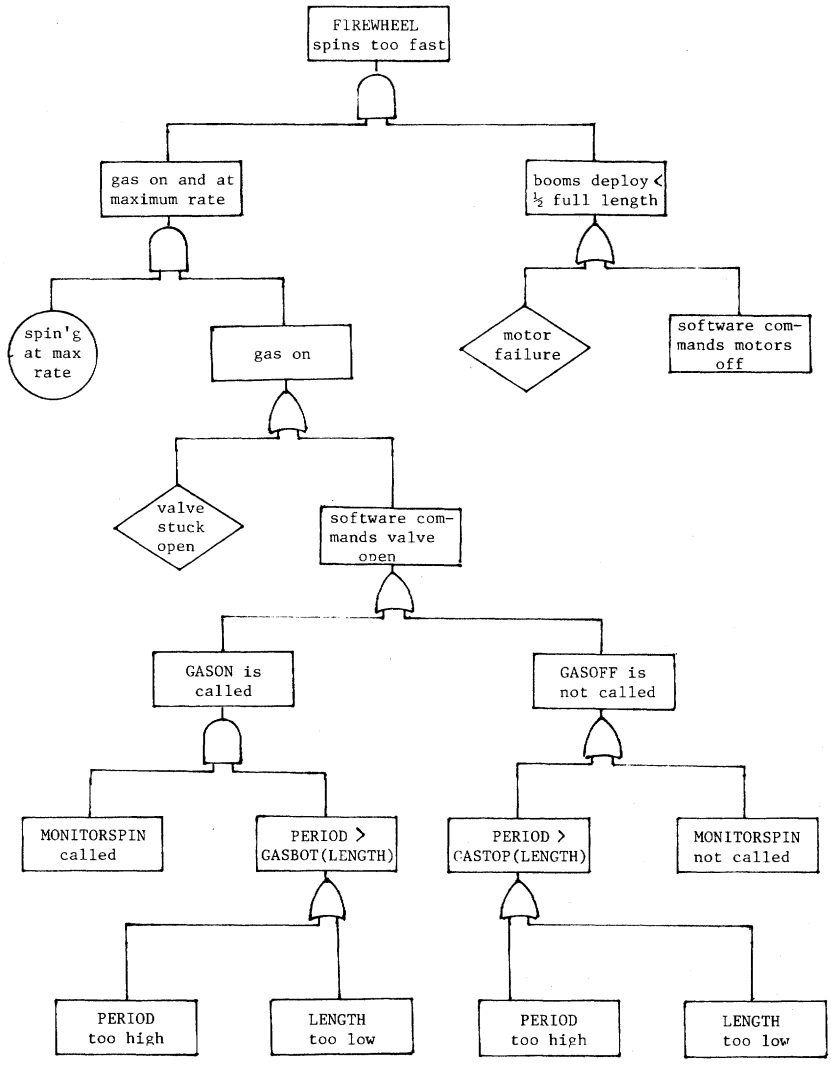


Fig. 7. FIREWHEEL spin control–Software opens gas valve.

predict in real-time systems, particularly when failures are expected to occur. This problem impacts the analysis by removing assumptions about order of processing. For example, in an interrupt driven system such as that described below, there are no assumptions made, for example, about which interrupt normally occurs before another, etc. Therefore, in the analysis if it is determined that a particular value for X is unsafe in routine F, then all places where X is set are analyzed for their ability to set X to that value. In effect, if a variable is ever set to a problematic value, then the system has a failure scenario and that scenario must be considered.

Experience with Software Fault Tree Analysis

Software fault tree analysis (SFTA) has been applied [8] to a program consisting of over 1250 lines of Intel 8080 assembly code which controls the flight and telemetry for a University of California, Berkeley, spacecraft designed for launch from a mother ship called FIREWHEEL (NASA/ESA). This is a real-time system of sufficient complexity to demonstrate the analytic power of fault trees in larger projects.

The mission of the spacecraft is to sample electric fields in the earth's magnetotail using wire booms deployed by the microprocessor after launch. Electrosensitive spheres at the end of these booms are continuously sampled by the microprocessor, which transmits the information in the telemetry format. Other instruments are controlled and readings taken by the same program. Due to cost considerations, there is no earth-satellite command control. Hence the microprocessor has to make all its own decisions.

The wire booms are deployed by centrifugal force applied to the spheres by the rotation of the spacecraft. Since the spacecraft naturally slows down when the booms increase in length, the microprocessor also controls a small jet which is pointed tangentially to the spacecraft cannister. Opening the gas valve increases the spin velocity. Mission objectives require that the spin rate be within a certain range of values.

During the deployment of the booms, it is posited that the spacecraft could fail such that the spin rate causes excessive centrifugal force on the spheres. Beyond 25 times gravity, the wires will be ripped off the spacecraft. It is known that early in the deployment, the force will be more than 20 times gravity. The critical failure event then is the destruction of the wire booms.

Figs. 7-9 illustrate the most important parts of the fault tree. Some detail was omitted for clarity. Relevant parts of the program code translated into pseudo-Pascal for readability are shown in the Appendix.

The loss event "FIREWHEEL spins too fast" occurs when the

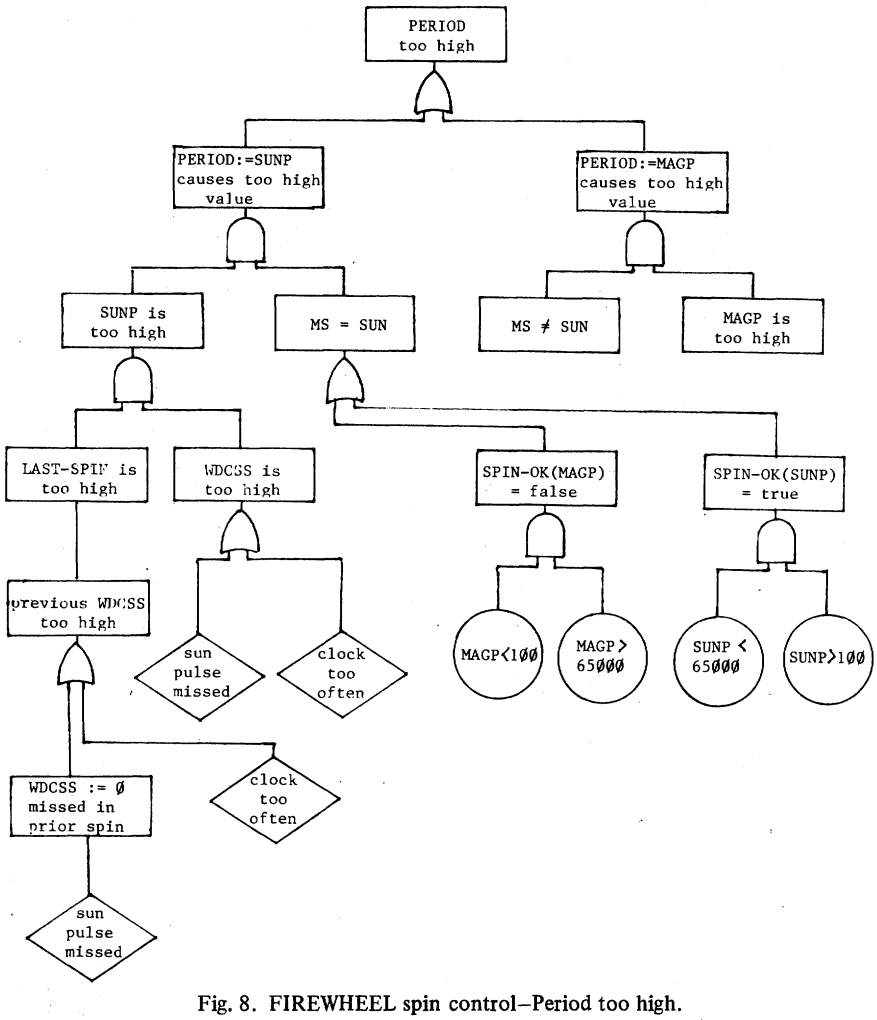


Fig. 8. FIREWHEEL spin control–Period too high.

spacecraft is spinning at the maximum rate, the gas is either turned on or left on, and the booms have stopped less than half way out. Past the half-way point, the gas has insufficient impulse to power the craft past the critical point (see Fig. 7).

Fig. 7 examines the ability of the system to open the gas valve. The software could at any time either open the valve, close it, or leave it as it was. Thus the "gas on" failure is true if GASON is called or GASOFF is not called when spinning at the maximum rate. GASON is called from the routine MONITORSPIN (see the Appendix). MONITORSPIN is a table-driven decision maker, and thus not much can go wrong except for erroneous input values. "Period > gasbot[length]," for example, is erroneous if either the period is too high or the length is too low. (GASBOT is a monotonically increasing function of length.) These conditions are AND'ed with the fact that MONITORSPIN must have been called in order to execute the statement. Similarly, GASOFF is not called when it should be if either MONITORSPIN is not called, or MONITORSPIN is called but period is too high or length is too low.

The possibilities of the period being too high and the length being too low are analyzed in Figs. 8 and 9, respectively. Relevant portions of the period algorithm appear in the Appendix. Basically, two mechanisms are able to supply the spacecraft period. One is a millisecond counter which is recorded and reset at each sun-pulse interrupt. The other is a sine wave fit of the magnetometer data, averaged over eight periods. A very weak assertion called SPINOK is applied to catch nonsensical values, but it is not likely that many problems will be found by this step.

One of the critical values for PERIOD is the sun period (the variable SUNP). Procedures RESTART3 and RESTART4 use the 1 ms telemetry interrupts for the clock. Each sun-pulse interrupt records the sun period as measured in the last rotation. Since missed sun pulses due to precession of the craft are thought to be likely, a minimum function is applied to this value and the value of the period before it. In this way, a single missing sun-pulse and the erroneously high period value will not result in the sun period value being too high. As shown in the fault tree, it takes two missing sun pulses to cause a high sun period value.

Fig. 9 explores the event in which the boom length is calculated too low. Since it is a simple sum of two values measured periodically in the telemetry formatting, either measurement is low if the sampling routine fails (not very likely) or the boom potentiometer fails. Finally, since the booms are lengthening, an indefinite loss of sampling will result in a low value.

The loss of sampling essentially involves a system crash, i.e., interrupts are disabled while the code is engaged in some infinite loop. Each of the interrupts cause an automatic interrupt-disable, but since none of these routines contain loops

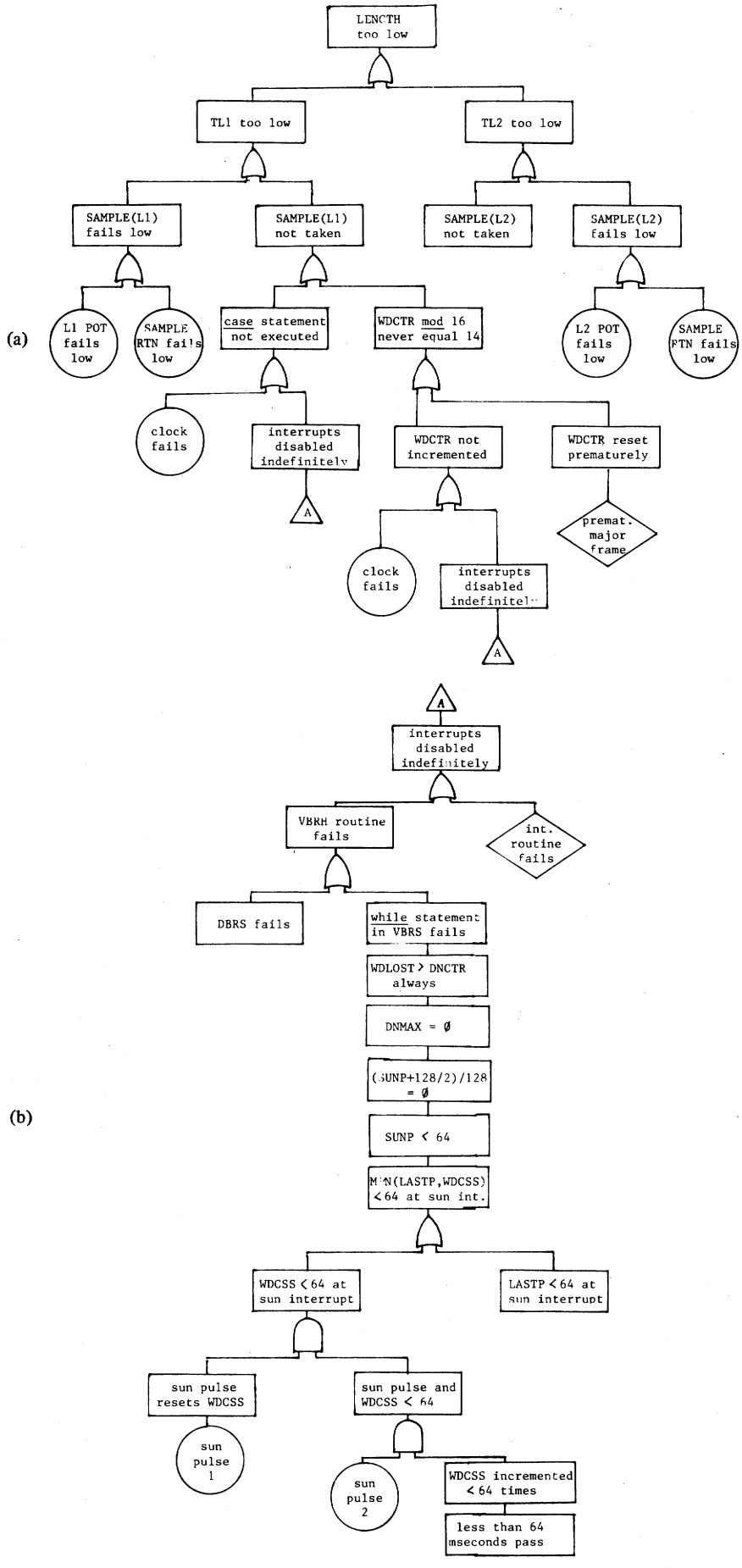


Fig. 9. (a) FIREWHEEL spin control–Boom length measured too low. (b) Boom length too low (continued).

or halts, we leave the node "interrupt routine fails" without further analysis. On the other hand, a routine called VBRH which is part of the asynchronous executive of the program, possesses the ability to disable interrupts. The routine performs a high rate sampling experiment which lasts 64 ms. For accuracy in the measurement, this routine must disable interrupts. Thus the VBRH routine must be analyzed for conditions which cause it to crash before it reenables interrupts.

For the high rate experiment, the sampling of data is easy, while repairing the side effects of an extended interrupt-disable is not. Variables which would normally have been updated by the continuous flow of telemetry interrupts must be modified to ameliorate the effects of the outage. In particular, the angle of the spacecraft with respect to the sun requires updating the word-count-since sun (WDCSS), the sun angle (THETA), and the phase-locked-loop down-counter (DNCTR). Since the microprocessor does not have a divide instruction, the while loop is used to perform that function. The deduction "DNMAX = 0" follows from the fact than an infinite loop in a while statement requires the condition to be always satisfied. In this case, "WDLOST > DNCTR" is always true. Although the intent of the loop is to modify WDLOST so the condition eventually becomes false, we deduce that WDLOST never changes by proposing a zero subtrahend (DNMAX).

DNMAX is set by the sun-interrupt routine RESTART3. The breakdown in logic sufficient to produce a zero in DNMAX is shown in the fault tree. Basically, two sun-pulse interrupts within 64 ms of each other are capable of crashing the microprocessor. Coupled with certain conditions of the deployment sequence, the interrupt pair is also capable of ripping off the booms (the original failure event for which the analysis was employed). And although it is nearly impossible for real sun pulses to occur so close together, one of them need only be a gamma ray induced spike, a common occurrence in space electronics. A simple check, or "blocking element," could have been inserted in the code to prevent this condition.

The fault tree analysis described above required a close look at about 12 percent of the FIREWHEEL program and two days of effort. By comparison, the analysis effort required to find the failure scenario through proofs or testing (if possible) would surely dwarf that which was needed for fault tree generation. In fact, before the fault tree analysis was performed, the program was thoroughly tested by an independent group which failed to reveal the 64 ms problem which the above analysis found.

Use of Software Fault Tree Analysis

Once a hardware fault tree is built, it can be converted to strings of Boolean expressions for easier analysis and then simplified using Boolean algebra. Each string then represents a failure scenario or set of circumstances which together can cause the loss event. Reliability estimates are assigned to the components and a failure statistic produced. Sensitivity analysis can be used to measure the effect of each event on the loss event statistic [2]. This numerical analysis determines the most likely causative agents of the loss event by observing which events in the tree make the most dramatic changes in the loss statistic.

It is theoretically possible to use reliability statistics for software components [15] in the same manner. There are some important assumptions, however, which must be made in applying these essentially hardware analysis techniques to software. The most important is independence of components. Hardware components usually fail independently whereas software component failures may be correlated. Although modern software engineering techniques strive to reduce this interdependence, it is not clear how meaningful the figures produced from such a numerical analysis would be. It might be possible to treat the loss expression as a relative statistic for the loss event and the results used for comparing alternative designs. But a great many caveats must be placed on the use of these numbers. Although more study is needed, fault tree analysis may prove useful for measuring software safety.

The most important uses of SFTA do not rely on this numerical analysis, however. First, the determination of the most likely causes of a loss event can be used to guide testing and to determine where relative effort must be placed to ensure safety. This information can also be used to determine which modules are the most critical and thus require special fault-tolerance procedures and/or formal proof of correctness.

Besides guiding testing and design, the conditions included in the failure scenarios can be used to identify unsafe states and the conditions under which fail-soft and fail-safe procedures should be called. This information can in turn be used to dictate the contents and placement of run-time assertions.

It may at first appear that run-time assertions, outside those on input values, will not be necessary since the analysis procedure can be used to verify that the software is "safe." That is, software errors leading to mishaps can be detected by the method and eliminated prior to operational use of the system. However, there are several reasons why this is not true (as it is, for example, with proof of correctness).

First, there are practical reasons why the entire code-level analysis may not be completed. For a large system, this may just not be possible. One positive aspect of this analysis procedure, however, is that at any level of abstraction the analysis may stop and run-time assertions inserted in the code to trap unsafe states. Expanding the entire tree down to the code level is not necessary for the analysis to be useful. In fact, only extremely critical parts of the software may need to be analyzed at the code level. SFTA helps to identify these critical areas.

Furthermore, some assertions cannot be checked prior to run-time. For example, real-time software has obvious time-critical elements. The results may be correct but of no use unless they are timely. Assertions to check for fail-safe time limits will be necessary. Also assertions which involve the state of parts of the system external to the logic of the software, e.g., the environment in which the software operates such as hardware or support systems, will be necessary. Environmental failures or interfacing problems cannot be prevented by a purely software analysis of safety.

Discussion

In this paper, three claims are being made.

1) There is a need to distinguish safety as an important quality of reliability.

2) Safety should be specified and verified separately from functional correctness.

3) Fault trees are a convenient method for accomplishing some of these goals.

Each of these points needs further elaboration.

First, one reason to separate out certain software features from functionality is to focus special attention on them. There are some software features, e.g., security and safety, which when violated assume great importance. Hardware fault tree analysis, for example, was originally developed jointly by Bell Labs, Boeing, and the Air Force to investigate the conditions which might cause the inadvertent launch of a Minuteman missile. In a nuclear power plant, the avoidance of radiation leakage assumes great importance when compared to other goals of the system (e.g., supplying power). One of the lessons learned in system safety engineering [7] is the importance of distinguishing safety as an important quality of the system and of providing responsibility for attainment of it. A major antiballistic missile system had to be replaced early because of serious safety accidents caused by previously unnoticed interface problems. Later analysis [18] suggests that the cause was the lack of identification and assignment of safety as a specific responsibility. Instead, safety was considered to be every designer and engineer's responsibility. Since that time, system safety has received more and more attention with strict standards being issued and enforced.

By distinguishing safety from other aspects of the function of the system, not only is attention guaranteed to be focused on safety, but it becomes possible to measure and ensure safety separately. As more and more safety critical systems become computerized, the government and private institutions are requiring that the builders of these systems demonstrate that their systems are safe. In order to accomplish this, techniques must be available which focus on safety separately from the general problem of correctness and from other software quality factors. With these types of systems, safety acceptance criteria are usually separated from reliability acceptance criteria.

It is possible to separate the states of a system into four categories:

1) correct and safe
2) correct and unsafe
3) incorrect and safe
4) incorrect and unsafe

using the usual definition of correctness and the definition of safety given earlier. As an example of each, consider an airplane navigation system where the hazard is defined as a collision. The plane is in a correct and safe state if it is headed for its desired destination and there is no danger of collision. The plane is in a correct but unsafe state if another plane is on a collision course with it, although perhaps quite far away. It may be necessary to temporarily go into an incorrect but safe state to avoid the collision. In this case, the goal of heading for the desired location must be subservient to that of safety. Thus the goals of a system can be divided in those of 1) attaining the mission and 2) what can and cannot happen while attempting to achieve the mission. The important point is that separation of the goals into two types allows for conflict resolution when all the goals cannot be met at one time or when one goal must be temporarily subservient to another. This is not, by the way, a unique notion. Lamport [10] includes both liveliness assertions (what the program must do) and safety assertions (what the program must never allow to happen) in specifying concurrent program modules.

Separate treatment of safety is also important because of the difficulty of writing complete functional specifications [6]. By specifying and verifying safety separately, a different viewpoint is employed, i.e., what should *not* happen versus what should happen. Although these are logically equivalent, a safety specification provides a separate and somewhat independent specification and verification of part of the system which may help to point out errors or omissions in the functional specification. Not only that, but the omissions and errors which are found will tend to be those with the most critical consequences. Thus, it is desirable to verify for hazards even if the system has been proved correct, especially for extremely safety critical functions.

Finally, safety specification and verification is a useful way of thinking about systems, i.e., it provides discipline and a way to look for errors and forces the system designer to think about what could go wrong. Harvey wrote the software for FIREWHEEL and was convinced all cases had been considered. The software was thoroughly tested by an independent group, and the attempt to apply software fault tree analysis to the program was originally thought to be just an exercise since the code was correct. Yet by turning his thinking around, Harvey was able to discover with two days work a possible problem with the system which he had not considered during the entire development of the system. In this case, proof of correctness would probably not have found the error because the problem arose from an unconsidered environmental condition. It is impossible, of course, to consider all possible environmental conditions in the formal validation of a system. But by focusing on safety, the most critical environmental conditions can be determined, in this case, two sun pulses within 64 ms of each other.

Even if the reader is now convinced that safety verification is possible and necessary, there still remains the question of whether fault trees are a useful tool to help accomplish this. It is possible to simply specify and verify safety assertions using standard reliability techniques as has been done for security. Fault trees have some advantages over standard verification procedures however.

First, fault trees provide the focus needed to give priority to catastrophic events and to determine the environmental conditions under which a correct (or incorrect) state becomes unsafe. For example, a brake failure in a car is incorrect but it is not unsafe unless the car is moving. As discussed above, dif-

ferent responses may be required under these two different conditions. Determination of these conditions may not be as obvious as in this example. The fact that two sun pulses within 64 milliseconds of each other could possibly destroy the spacecraft was never recognized while the FIREWHEEL system was being developed or tested. It is important to recognize, however, that even with the use of safety analysis techniques, such as fault trees, the consequences of what turn out to be critical environmental conditions may not be recognized or understood until it is too late. No analysis technique can guarantee completeness. However, it is hoped that focusing on safety will help minimize the number of such undetected and unhandled hazards. Furthermore, analysis techniques *must* be combined with run-time safety techniques [13], [19] to attempt to detect and recover from unsafe states resulting from software errors or critical environmental conditions which are not recognized prior to operational use of the software.

A further advantage of fault trees is that they provide a convenient structure to store the information gathered during the analysis which can be used later for redesign (e.g., adding redundant software routines), testing, measurement, inclusion of run-time safety assertions and exception-handling procedures, etc. Also the technique and structure is already familiar to hardware engineers. Safety is a system quality and must be analyzed from a system viewpoint. Fault trees provide one single structure for specifying software, hardware, human actions and interfaces with the system, etc. Thus the entire "system" can be analyzed as a whole and the appropriate measurements for reliability and safety derived.

Finally, whereas in verification underlying machine functions are assumed to operate "correctly," i.e., according to the specifications, fault-tree analysis allows the examination of the effects of underlying machine failures or environmental changes. Although theoretically it should be possible to verify all underlying and associated systems, this is not usually practical. For example, although great advances have been made in hardware fault tolerance, it is not possible to guarantee the complete absence of hardware failure.

The applicability of fault tree analysis, however, is very narrow and care must be taken to use it only in appropriate circumstances. First, and most important, fault trees are useful only for safety analysis and should *not* be used as a general reliability technique. That is, to start with the loss event "system fails" is hopeless. For fault tree analysis to be practical and useful, there must be a small number of critical failures which can be distinguished from the myriad other possible failures by their catastrophic consequences. If this is not the case, i.e., the number of safety critical failures is very large (which may imply a need for redesign of the system) or the system is such that the consequences of some failures are not catastrophic with respect to other failures, then regular proof of correctness methods should be used without fault tree analysis.

More experience with using fault tree analysis on real systems is necessary to determine where, when, and if the technique is useful and practical. Accordingly, a software tool is being developed by the Irvine Safety Project which will aid in the production and analysis of fault trees. The tool will enable us to gain this experience and to develop and test heuristics and techniques which will reduce the amount of effort required in doing the analysis.

Appendix

Relevant parts of the FIREWHEEL program written in pseudo-Pascal follow.

```
function PERIOD:integer;  /* redundant routines for
                             calculating spin period */
    begin
       if SPINOK (SUNP)          /* determine which of MAG or */
           then MS:= SUN         /* sun info is working OK    */
           else if SPINOK (MAGP)
                 then MS := MAG
                 else MS := SUN  /* use sun if neither */

       if MS = SUN
           then PERIOD := SUNP   /* return sun-period or */
           else PERIOD := MAGP   /* mag period depending */
    end


function SPINOK (PER:integer):boolean;
       SPINOK := PER > 100 and PER < 65000;


function LENGTH:integer;          /* returns tip-to-tip length
                                     of boom wires             */
       LENGTH := max (TL1,0) + max (TL2,0);


procedure RESTART4;               /* telemetry interrupt (milsec) */
    begin
       if DNCTR <> 1                /* count down until 0. Then   */
          then DNCTR := DNCTR - 1   /* reload the count and bump  */
          else begin                /* the sun angle theta        */
                  DNCTR := DNMAX;
                  THETA := THETA + 1 mod
               end;
       WDCSS := WDCSS + 1;          /* record msecs since sun     */
       WDCTR := WDCTR + 1;          /* and update T/M word number */
       case WDCTR mod 16 of         /* do one of 16 routines      */
           0: ...
           .   .
           .   .
           .   .
          14:  TL1 := SAMPLE(L1) /* sample boom one length pot */
          15:  TL2 := SAMPLE(L2) /* sample boom two length pot */
       end;
    end



procedure RESTART3;               /* sun pulse interrupt */
    begin
       SUNP := min(LASTP, WDCSS) /* sun period is min of
                                    the last two periods */

       /* Round off and divide period into 128 parts.  Each is
          the number of milliseconds for one pseudo-sun-degree */

       DNMAX := min((SUNP+64)/128,255); /* set counter maximum */
       DNCTR := DNMAX;                  /* and the counter too */
       THETA := 0;                      /* reset sun angle     */
       LASTP := WCDSS;                  /* record last period  */
       WCDSS := 0;                      /* reset words since sun */
    end


procedure VBRE;          /* high rate sampling of spheres */
    begin
       disableints;                     /* disable 8080 interrupts */
       for I := 0 to 127 do             /* and sample 128 points   */
           RAM[I] := DBRS;

     /* Because of the interrupt disable, update vital
        variables such as the no. of words since sun, the
        sun angle and the phase-locked-loop down counter */

       WDLOST := 64;                    /* words lost during disable */
       WDCSS := WDCSS + WDLOST;         /* fix words since sun pulse */

       While WDLOST >= DNCTR do         /* update sun angle */
           begin
               WDLOST := WDLOST - DNCTR;
               DNCTR  := DNMAX;
               THETA  := THETA + 1 mod 128;
           end;
       DNCTR := DNCTR - WDLOST          /* set new down count */
       enableints;
    end


procedure MONITORSPIN;    /* check spin; manipulate gas and
                             motors during deployment       */
    begin
       P := min (PERIOD/64, 255);
       L := min (LENGTH/16, 15);
       if  P < GASTOP[L] then GASOFF;
       if  P > GASBOT[L] then GASON;
       if  P > BOOMTOP[L] then MOTOROFF
             else if  P < BOOMBOT[L] then MOTORON
    end
```

Acknowledgment

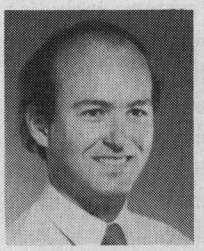
The authors would like to thank the editor and the referees along with R. London, P. Lee, D. Berry, and the members of the Irvine Safety Project, especially T. Shimeall, J. Stolzy, and J. Thomas, for their helpful comments and discussion on earlier drafts of this paper.

References

[1] K. R. Apt, "Ten years of Hoare's logic: A survey," *ACM TOPLAS*, vol. 3, pp. 431–483, Oct. 1981.
[2] R. L Browning, *The Loss Rate Concept in Safety Engineering*. New York: Marcel Dekker, 1980.
[3] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," presented at the 8th Int. Conf. on Fault-Tolerant Computing, Toulouse, France, June 1978.
[4] R. Dunn and R. Ulman, *Quality Assurance for Computer Software*. New York: McGraw-Hill 1982.
[5] J. R. Garman, "The bug heard 'round the world," *Software Eng. Notes*, vol. 6, Oct. 1981.
[6] S. L. Gerhart and L. Yelowitz, "Observations on the fallibility in applications of modern programming methodologies," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 195–207, Mar. 1976.
[7] W. Hammer, *Handbook of System and Product Safety*. Englewood Cliffs, NJ: Prentice-Hall, 1972.
[8] P. Harvey, "Fault-tree analysis of software," Master's thesis, Univ. California, Irvine, Jan. 1982.
[9] C.A.R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576–580, Oct. 1969.
[10] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 125–143, Mar. 1977.
[11] N. G. Leveson, "Software safety: A definition and some preliminary ideas," Univ. California, Irvine, Tech. Rep. 174, Mar. 1981.
[12] —, "Software safety from a software viewpoint," in *Proc. 5th Int. Syst. Safety Conf.*, vol. 2, July 1981.
[13] N. G. Leveson, T. Shimeall, J. Stolzy, and J. Thomas, "Design for safe software," in *Proc. AIAA Space Sci. Meeting*, Reno, NV, Jan. 1983.
[14] M. Lipow, "Prediction of software errors," *J. Syst. Software*, vol. 1, pp. 71–75, 1979.
[15] C. V. Ramamoorthy and F. B. Bastiani, "Software reliability—Status and perspectives," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 354–371, July 1982.
[16] C. V. Ramamoorthy, F. B. Bastiani, J. M. Favaro, Y. R. Mok, C. W. Nam, and K. Suzuki, "A systematic approach to the development and validation of critical software for nuclear power plants," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 231–240.
[17] B. Randell, "System structure for software fault tolerance," in *Proc. Int. Conf. Reliable Software, SIGPLAN Notices*, vol. 10, no. 6, June 1975.
[18] W. P. Rodgers, *Introduction to System Safety Engineering*. New York: Wiley, 1971.
[19] J. Thomas and N. G. Leveson, "Applying safety design techniques to software safety," in *Proc. AIAA Space Sci. Meeting*, Reno, NV, Jan. 1983.

Nancy G. Leveson received the B.A. degree in mathematics, the M.S. degree in management, and the Ph.D. degree in computer science from the University of California, Los Angeles.

She has worked for IBM and is currently an Assistant Professor of Computer Science at the University of California, Irvine. She has published articles on software fault tolerance and safety and in 1980 started the Irvine Safety Project, a group engaged in the study of safety-critical real-time systems. The Safety Project currently has seven graduate students working on such topics as software fault tree analysis, specification and design of safety critical software, run-time safety facilities, management of safety critical software projects, and the use of Ada in fault-tolerant and safety-critical applications. She has also published in the area of centralized and distributed database design and semantic integrity.



Peter R. Harvey received the B.A. degree in computer science from the University of California, Berkeley, and the M.S. degree from the University of California, Irvine.

He worked for the Space Sciences Laboratory for three years and was involved in the design and programming of scientific satellites. He directed the computer facility at the laboratory until leaving to continue his education and has since returned to work at the Space Science Laboratory. He is currently designing three instruments for NASA spacecraft and for the U.S. Air Force CRRES satellite.