Information and Computer Science

Applying Existing Safety Design Techniques To Software Safety

by

Jeffrey C. Thomas

and

Dr. Nancy G. Leveson

TR#180

UNIVERSITY OF CALIFORNIA
IRVINE

Applying Existing Safety Design Techniques To Software Safety

by

Jeffrey C. Thomas

and

Dr. Nancy G. Leveson

September 1981

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

This work was supported in part by a contract from Hughes Aircraft Co. (7-656146-T-DS).

1.0 INTRODUCTION

Recently software safety (the measure of the ability of a system to avoid safety failures which are the result of software errors) has assumed a position of importance within the general area of system safety [GRI81]. Many of the techniques applied to hardware by safety engineers are applicable to software. However, because of the differences between hardware and software, a simple transfer of technology is not possible. Software does not "wear out", but errors are a result of the complexity of the software design and implementation process. Further, software errors remain even after extensive testing.

This paper will attempt to survey some of the existing hardware and software techniques which can enhance software safety and to suggest how these techniques can be implemented in software. This particular investigation will be concerned mainly with software design techniques which enhance the overall safety of the system. It will be assumed that safe software design is only one part of the overall safety plan. In particular, it will be assumed that, prior to the software design phase, preliminary hazard analysis has been done on the system which the software is to control. Obviously, the software failure modes which can lead to safety failures are related to the system being controlled by the software.

Furthermore, particular importance will be placed on the applicability of current system safety techniques to software. It is important that software safety and system safety techniques cooperate as closely as possible. Ideally, software safety techniques should be able to fit naturally into the entire system safety plan.

The paper concludes with a summary of some general design principles for safe software. Although many of these techniques are not new, they have been applied in the past in an ad hoc and machine specific manner. Applied in a consistent and coherent manner, they can greatly enhance the safety of real-time critical systems.

2.0 DEFINITIONS

Communication problems are exacerbated by a lack of common definitions of terms. For the purpose of this paper, the following definitions will be used. Where hardware and software definitions of the same terms differ, the proposed standard definitions of the IEEE Standards Committee [IEE79] will be followed.

Hazard - a condition with the potential for causing loss of life or property.

Software Error - a human action or inaction (during development or maintenance) which results in software containing a fault.

Software Fault - a manifestation of an error in software. A fault, if encountered, may cause a failure.

Failure - The inability of a system or system component to perform a required function within specified limits. A software failure occurs when the failure is due to a software fault.

Reliability - the probability that a system, including all hardware and software subsystems, will perform a required task or mission for a specified time in a specified environment.

Safety Failure or Mishap - a failure which leads to casualties or serious consequences. A serious consequence is any undesired event which the designer considers to be as or more important than the correct (reliable) operation of the system.

Safety - the ability of the system to avoid safety failures.

Software Safety - the ability of the software system to avoid safety failures caused by software errors.

Unsafe State - one for which there are circumstances where further processing will lead to a safety failure.

Safe System - one which prevents unsafe states from causing safety failures.

Fail-Safe System - a system which limits the amount of damage caused by a fault. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

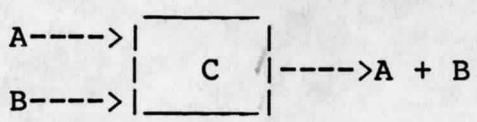
3.0 SOFTWARE FAILURE MODES

In considering designs to increase software safety, it is important to understand the dynamics underlying software failure.

There seems to be some confusion as to whether or not software fails. Some authors have claimed that software cannot fail [ERI81, GRI81]. Their argument is that software

faults are the result of errors introduced by humans during the creation of the software and do not arise spontaneously during execution (as is the case with hardware). Therefore they conclude that software does not fail and thus does not have failure modes. The problem with this reasoning is a confusion between cause and effect. A failure occurs when the software or hardware does not perform as expected. The antecedents of the failure may be important for post-failure analysis and repair but are not important with respect to whether the failure, caused by incorrect functioning, has occurred.

As an example, consider the following hardware box with inputs A and B and output A + B (logical or).



C can be said to have failed when the output is not A + B. This black box has several failure modes, including no output at all. Now if box C is replaced by a software routine to compute the same result, i.e. A + B, the failure modes remain the same. It is the failure to produce the correct output which causes the hazard. The definition of failure then is external to the black box and cannot depend upon whether the failure was the result of hardware or software problems within the box.

According to Hammer, systems failure mode and effect analysis (FMEA) considers four failure modes:

1. Premature operation of a component (operation not required [error of commission] or too early)
2. Failure of a component to operate at a prescribed time (operation left out of sequence [error of omission] or too late)
3. Failure of a component to cease operation at a prescribed time (calculation takes too long or no termination condition [infinite loop])
4. Failure of a component during operation, i.e. incorrect output.

It is clear that software components can exhibit each of these failure modes. That is, a module can be called at the wrong time or not at all, it can go into an infinite loop or take too long to execute, and it can produce the wrong output.

One important thing to note is that failure of a component to recognize a hazardous condition requiring corrective action would also be considered as a failure of the component during its operation.

In summary, there seems to be little doubt that software can fail and has identifiable failure modes.

4.0 SYSTEM SAFETY TECHNIQUES

In system safety five basic techniques are used to enhance design safety. These are hazard elimination, hazard limitation, lockouts, lockins, and interlocks, fail-safe design, and failure minimization. The applicability of these techniques to software will be examined in turn.

4.1 HAZARD ELIMINATION

During the design phase of a system, when a hazard is discovered, elimination of the hazard should first be attempted. It could be argued that it is not possible to eliminate the hazard through software because a hazard is very hardware or environment dependent. For example an airplane has the hazard of crashing because it is in the air. The only way to remove the hazard is to get the plane on the ground, and this cannot be accomplished exclusively through the software. The key word is "exclusively". Software may be used in conjunction with the hardware to help eliminate the hazard, in this case, to issue the commands to the hardware which are necessary to land the plane.

Just as software and hardware have similar failure modes, as argued above, hazards need not be limited to hardware. In the terms defined above, a hazard was a

condition which could lead to a mishap or safety failure. In software terms, this is an unsafe state. Further a hazard or unsafe state is the result of a critical fault which in turn stems from a critical error. Thus to eliminate hazards in software, it is necessary to eliminate critical errors or critical faults.

Eliminating all critical errors is probably an unrealistic goal. But many techniques have been developed in software engineering and system safety to aid in this process. These techniques on the software engineering side include formal requirements and specification languages, design techniques and tools, and project management techniques. System safety techniques include preliminary hazard analysis (PHA), failure modes and effects analysis (FMEA), fault tree analysis (FTA), etc.

Some critical faults can be eliminated through program verification and validation techniques and tools, including program debugging, automated testing, independent test and evaluation, and proof of correctness. It may also be possible to minimize the number of remaining critical faults by changing as many as possible to non-critical faults, that is, to design such that non-critical functions are separated from critical functions and such that non-critical modules are in turn prevented from causing a safety failure. For example, non-critical modules might be prevented from

accessing critical resources or the system might be designed so that an error in a non-critical module cannot bring down the entire system. This latter might involve the use of watchdog timers, dynamic system reconfiguration, and strict independence of modules.

4.2 HAZARD LEVEL LIMITATION

In many instances in system safety the hazards cannot be eliminated but the level of the hazard may be reduced. For example, there may be certain parameters (radiation, temperature, pressure) that can be monitored for their safety level. Monitoring could be used to keep track of safety critical parameters. If these parameters exceed certain levels, the monitoring program could invoke a procedure that returns the parameter to a safe level. For example, in an intensive care unit, if the level of oxygen was too low the monitor could react and activate oxygen pumps in the area. The key in hazard level limitation is to detect the hazardous condition at a level low enough to ensure enough time to take action. If the hazard level became too high the monitoring program could have the option to fail-safe.

Monitors can be employed to indicate [HAM72]:

1. whether or not a specific condition exists
2. whether the system is ready for operation or is operating satisfactorily as programmed
3. if the required input is being provided
4. if the output is being generated
5. if the limit is being met or exceeded
6. whether the measured parameter is abnormal.

These monitoring functions may be implemented in software.

Monitoring by itself is of no use. It must be coupled with a means of applying corrective action. The four principal steps in this process are detection, measurement, interpretation, and response.

Detection must be very sensitive and capable of sensing specific parameters. There are several ways of sensing: continuously, continually but intermittently, or intermittently at the desire of an operator or procedure. The method used will depend on the rate of change of the parameter and its importance with respect to safety. It is very important to be able to detect abnormal conditions at as low levels as possible to permit corrective actions to be taken in time.

Measurement and interpretation are closely related. There are different ways of measuring parameters, and it is the purpose of interpretation to make clear what the measurements mean so that operators or programs can take appropriate action. If the information taken is being displayed to a human operator, it is very important that it be done in a concise and useful manner. Studies are currently being done to aid in design of systems with human interaction [ROU81]. Models have been proposed for different types of human-computer interactions. Some models can be used to determine the variables to be displayed and the appropriate format for the display. Some of the techniques have even been used to evaluate alternate instrument display formats for the Boeing 737. In the study of the Boeing 737, a pictorial display was determined to have many advantages over the more traditional display.

In normal situations no response is required; however, when an abnormal situation is detected responsible operators must be notified quickly. After notification, either recovery or fail-safe measures will be taken.

When designing monitoring programs it is important to note several things. Monitors themselves must be highly reliable and the checking and cross-checking of the monitor may be very critical. Monitors should be independent so that a failure of a part of the program will not cause a

failure in the monitor. Also the system should be designed so that a failure of the monitor will not cause a failure of the entire program. Along with monitoring parameters, critical actions might be monitored to ensure correct performance (buddy system).

Besides monitoring hardware, monitors can be used to detect unsafe states in the software. This might include using assertions to check important parameters. It is important to detect hazardous conditions, i.e. unsafe states, while there is enough time to prevent a safety failure. Thus the earlier a fault is found the better, and continual or intermittent monitoring is important.

Warnings can be used in conjunction with monitoring programs. Warnings serve to focus attention on the hazard. Warnings may be sent to human monitors or may be in the form of messages sent to other procedures to point out abnormal conditions. The warnings would be used to avoid a potentially dangerous action or to initiate fail-safe procedures.

One further way to limit hazards in software is to minimize the amount of time an unsafe state exists. For example, a program might not send the command to arm a missile until it is near its target. This might reduce the hazard of the missile being detonated too close to its

launch site or having the shutdown or fail-safe procedures fail. In general, this example illustrates the principle of starting out in a safe state and later having the software signal a switch to a possibly unsafe state instead of starting in an unsafe state(e.g. armed for detonation) and later possibly switching to a safe state (e.g. shutdown) if a hazard is detected.

4.3 LOCKOUTS, LOCKINS, AND INTERLOCKS

Lockouts, lockins, and interlocks are based on two principles:[HAM72]

1. isolating a hazard once it has been recognized
2. preventing incompatible events from occurring, from occurring at the wrong time, or from occurring in the wrong sequence.

Methods to deal with these problems can improve software safety since many software failures are caused by omission, commission or timing errors. One type of software error that has not been studied in detail is that of commission (doing what is not required). Software has been very intensively tested to make sure that it does what it is specified to do but due to its complexity it may be able to do a great deal more. Means to limit the actions of the software are needed.

A lockout prevents an event from occurring or prevents something from entering an unsafe state. A lockin maintains an event or keeps something from leaving a safe state. One way that operating systems have incorporated these concepts is in the use of capabilities [DVH66].

Capabilities have been proposed to handle the problem of shared files and memory. Essentially a capability is a permission to use certain objects or procedures. It can be implemented in the form of a list of permissible actions associated with each process (program in execution). The main principle here is that each process should have no capability beyond what is required to perform its task. Capabilities must be explicitly given, so that the default is to have no access. This could be useful in real-time critical applications by allowing only routines with the proper authorization to access safety critical modules. An example might be that only a few routines would have access to a routine that launches a missile. Using capabilities would rule out other routines accessing the launch routine by a mistake in coding.

Interlocks are mechanisms provided to avoid timing failures. They prevent events from occurring inadvertently by requiring a preliminary action to have taken place first. Again to prevent an unexpected missile launch the system might be designed so that two procedures must be activated

in sequence. The first might be a procedure to ready the launch and the second actually would be the command to launch. The second would need the information that the first had already been activated before it would continue. This could also eliminate mistakes due to the accidental call of a single critical procedure. If there are two conditions or actions that should not take place together, an interlock can be used to isolate the two events in time. Guard gates at a railroad crossing keep pedestrians from the track when a train is near. After the train has passed the pedestrians can continue. In like manner checks can be put into procedures to determine if it is safe to continue. If one critical function is not occurring then others can proceed.

When the sequence of events is critical, an interlock can be used to ensure the correct sequencing. In some situations certain valves must be opened before filling a tank. If the pumping begins first, the tank may become overstressed by the unrelieved pressure. An addition can be made to modules to check whether the prerequisite actions have already taken place and then to signal when the resulting actions have been accomplished.

Many mechanisms along these lines have been developed for computer operating systems [SHA74] to deal with the problems of concurrency and synchronization. Mechanisms

which have been proposed to deal with these problems include semaphores [DIJ68], monitors [HOA74], and kernels [WUL75]. Real-time critical applications software in many respects has more in common with operating systems than with other types of applications such as business programs. Thus it seems reasonable that many of the techniques used in designing operating systems are more applicable for critical applications than the techniques used to design more pedestrian applications software.

4.4 FAIL-SAFE DESIGN

To date one area of software safety that has not received much systematic attention is that of fail-safe design. In hardware systems safety, fail-safe design is recognized as important because of the recognition of the inevitability of hardware failure due to fatigue. However,there has been an optimistic outlook in software for a long time that if only the proper design techniques can be developed, software design errors could eventually be removed. Recently, some software engineers are taking a more sober look and realizing that software is probably much too complex to really conquer using purely fault-avoidance techniques [AVI75]. Wulf has said that it is much more important to be able to recover from failures than to prevent them [WUL75]. Fault-tolerance has begun to play a bigger role in design, yet this alone is still inadequate.

Fault-tolerant techniques may fail and provision must be made to handle these failures.

Fail-safe designs try to ensure that the occurrence of a failure will leave the system unaffected or that the system can be put into a state in which no damage or injury could result. Fail-safe design can be categorized into three types [HAM72]:

1. fail-passive
2. fail-active
3. fail-operational

The goal of a fail-passive design is to reduce a system to its lowest energy level in the event of a failure. An example of a fail-passive design is a fuse for the protection of electrical circuits. When the system is overloaded, the fuse blows and inactivates the system, thus preventing further damage. It is possible to write such "software fuses" that disable parts of a system when a hazardous condition is detected. This has been used for some time in systems with a very large energy content such as missiles and nuclear generating stations, but their use has not been systematic or thorough. This should be an important component of all software safety systems.

There are some systems that you may not be able simply to "turn off" in the event of a failure. To maintain safety the damaged system itself must be kept in its energized condition, though it may continue operating with reduced function. Maintaining this situation is the goal of fail-active (fail-soft) design. Fail-active components may also initiate measures to eliminate the possibility of an accident caused by a failure. With a good understanding of the basic system, fail-active requirements can also be designed into software.

In the past, the simplicity of the systems has caused a tendency towards fail-passive design, but because of today's great dependence on software in many life sustaining situations, fail-active design may be of more importance. In the design of fail-active systems the software must be carefully divided into modules that can be reconfigured after a failure is detected. This will help to avoid a total "crash" of the system.

The most desirable design is one that allows the system to function fully and safely until corrective action is possible. Hammer calls this fail-operational and is closely related to fault-tolerant design,i.e. trying to keep the total system going at all costs. Fault-tolerance will be discussed in detail in the section on redundancy.

It is important to note at this point that these issues of fail-safe design are what cause software safety to differ significantly from reliability. Fail-passive and fail-active designs really do not add much, if anything, to the reliability of a system (i.e. the accomplishment of its mission), and in some practical cases it may reduce the total reliability of the system. One example might be a nuclear generator whose software has been making some crucial calculations. The calculations result in an answer that makes no sense and the only alternative to keep the generator going may be to try another routine to do the calculations by a different method. At this point the fail-safe monitoring program may come in and determine there is not enough time to redo the calculation before a safety critical point is reached so the safety monitor initiates the reactor shutdown routines to avoid catastrophe. This raises another issue in which software safety differs from reliability in that the system may be functioning perfectly, yet a safety monitoring program may detect an external condition that is unsafe and invoke safety shutdown. Here no reliability technique could be used to increase the safety. Reliability (redundancy) alone is not adequate but redundancy used together with fail-safe design may be very important in the trade-offs between performance and safety.

4.5 FAILURE MINIMIZATION

There are some systems that are so critical that even a fail-safe arrangement is less preferable than a system that will fail only rarely. For these systems, failure minimization will be the guiding design principle. In an air traffic control system a fail-safe action may be to stop all departing aircraft and to divert all arriving aircraft to other operational airports. One can see that this could be a very expensive action (passenger inconvenience) and therefore should not happen often. A combination of failure minimization and fail-safe techniques is therefore required.

Failure rate reduction methods try to limit failure during system operation. This might not seem to be applicable to software because software does not "wear out" as do hardware components. This is true for some hardware methods such as derating (using components whose capacity is greater than required to lessen wear out) and timed replacements (replacing components before they wear out). Screening may be partially applicable by means of extensive testing. Redundancy also would seem to have no usefulness in software, but when modified it can be applied in several ways. In fact most (if not all) work on fault-tolerant software has been concerned with various forms of redundancy.

In hardware there are two general types of redundancy employed - parallel redundancy and switching redundancy. First it is necessary to see how these are used in hardware and then how they have been modified for software.

In parallel redundant design, several components perform the same function at the same time. This is also known as masking redundancy since the multiple components are used to mask the errors of any one component by looking at the results of all components and determining a correct value. The masking of the fault thus occurs instantly and automatically. There are several methods for determining the resulting value (median select or voting). This type of redundancy has been used in systems requiring continuous operations within a specific time period. In hardware this has mainly been implemented through replication of individual electrical components or triple modular redundancy (TMR). An example of the latter is the computer to control the Saturn V, which was divided into seven parts, each part using TMR [CB71].

Switching redundancy uses standby units which are ready to take over when an operating unit fails. The key points of this design are the failure detection and the switchover (with recovery) to standby units. Examples of this method are the telephone's electronic switching system (ESS) and the self testing and repair (STAR) computer [AVI75].

Hardware redundancy techniques cannot be applied without some changes. The primary problem is that it does no good to have several copies of the same software running redundantly, because all copies will fail at the same place. Software failures are due to design errors and not component wear out. Thus the failures are deterministic and not random. Overcoming this problem requires several different implementations of a given specification. These implementations can be run in parallel to mask failures. Of course, if the specification is ambiguous or contains errors, redundancy still will not help. Before comparing the two methods of redundancy it would be helpful to see how they have been applied to software.

Algirdas Avizienis and his colleagues at UCLA have been the main proponents of using parallel redundancy in software (N-version programming). In [CA78], N-version programming is defined as "the independent generations of N>= 2 functionally equivalent programs, called 'versions', from the same initial specification. Chen and Avizienis state that this specification should be in a formal specification language and should define:

1. the function to be implemented by an N-version unit
2. data formats for special mechanisms: comparison vectors (c-vectors), comparison status indicators (cs-indicators), synchronization mechanisms.
3. the cross-check points (cc-points) for c-vector generation

4. the comparison algorithm
5. the response to the possible outcomes of comparison.

After specification the individual units are developed.

Several special mechanisms are required for N-version programming. Cross-check points (cc-points) are places in the programs where c-vectors are generated. After generation, the c-vectors are used in the comparison algorithms. Contained in the c-vectors are comparison variables and status flags which indicate if results matched and what resulting actions should be taken. Some possible actions are:

1. continuation
2. termination of one or more versions
3. continuation after update of some c-vectors.

When cc-points have been reached by a unit, synchronization mechanisms are used to signal the driver that the calculation is completed and the c-vector is ready for comparison. This eliminates the problem of comparison before all versions are finished. A supervisor program (driver) is also needed to coordinate the execution of the N versions and to compare their results.

Brian Randell and his colleagues at the University of Newcastle upon Tyne have applied switching or standby redundancy to software in what they call "recovery blocks" [RAN75].

A recovery block consists of a regular programming language block (called the primary block), an acceptance test, and a sequence of alternate blocks. The acceptance test is a logical expression which is evaluated to determine if the result of a block is correct. If a primary or alternate block does not complete (because of an error or expiration of time limit) or fails the acceptance test, the state is restored to that just prior to entering the recovery block, and the next alternate (if there is one) is entered. If all alternates fail to pass the acceptance test, recovery is attempted at the level of the next enclosing recovery block. If the acceptance test is passed, control is passed to the statement after the recovery block. Prior states to be used for rollback are stored in a "recovery cache".

The recovery block scheme gives general solutions to the problems of switching to the spare component and structuring the software system so that the extra software does not add to the overall complexity of the system. One difference of recovery blocks from hardware standby sparing is that once a procedure fails, it is not permanently

disabled. Instead the failure is treated as a transient fault as in hardware, so that the succeeding times the recovery block is called the sequence of modules to use is the same. This seems to be an unfortunate choice for software since design errors are relatively permanent without human intervention. For recovery blocks which are repeatedly executed much wasted time may result. Even worse, the assertion testing in recovery blocks is fallible and errors may sneak through with repeated execution of an erroneous recovery block. An alternative which avoids this problem will be described below.

In comparing the N-version programming and recovery blocks, it is helpful to look at the primary weak points of each. The main limitation of N-version programming is that all versions originate from the same specification. Besides not being able to determine if a specification is unambiguous, it is very difficult to tell if one is complete. Recent experience in software engineering has 60% of software errors due to an inadequate specification [LIP79]. Also if a problem has multiple distinct intermediate or final solutions, there would be no way to compare the different versions correctly.

The limitations of the recovery block involve the use of acceptance tests and the restoration of the system state after a faulty block. It may be very hard to design an

acceptance test that is complete, and Randell admits that in practice an incomplete test might have to suffice due to cost and complexity [RAN75]. The problem of restoring the system state is one of overhead. All values that are reset during a block must be saved until the block has been accepted, otherwise the initial values must be restored. To handle this efficiently, special hardware will be needed. There have been some experiments with a hardware recovery cache [RAN75].

N-version programming, because of its parallelism, would be more time efficient than recovery blocks. N-version programming's error detection seems to be simpler and more complete because an error is signalled whenever the versions do not match. This offers much broader coverage than the acceptance test in recovery blocks. Recovery blocks do seem to have a finer application than N-version programming. N-version programming has usually been employed at a grosser level (replication of entire programs) than at the block level, causing much more program development costs than necessary. However, Chen and Avizienis have experimented using N-version programming at a finer level [CA78]. Recovery blocks may also be a little less expensive to develop since the alternate version do not need to be completely independent. This opens the way to use older (perhaps more stable) versions of programs to back up newer versions.

Overall there are many tradeoffs between these methods. The particular application has to be carefully considered before one method is chosen. In the future most benefit would seem to come from providing redundancy only at the points that are safety critical and by providing that type of fault-tolerance which is most appropriate for the particular critical function being handled.

Because of the many disadvantages of N-version programming and recovery blocks, a new type of mechanism to support software redundancy should be devised. A preliminary study of this problem has determined several requirements and proposed some solutions.

In considering the addition of any mechanism to increase safety, safety engineers must carefully evaluate any added complexity. Any increase in complexity usually has a harmful effect on safety, so one goal in adding a new mechanism would be to increase the simplicity of the software design as a whole. Also the more simple the mechanism is, the easier it is to modify if there are changes in requirements.

Closely related to the goal of simplicity is that of reliability. If the mechanisms supporting software redundancy are themselves unreliable, then the advantages of redundancy are lost. It would be beneficial if this

redundancy mechanism could be standardized and be used in many software safety programs. Finally, if the responsibility for fail-safe decisions can be centralized, this would increase the understandability of the software.

To respond to some of the above goals, a safety executive kernel has been suggested. The general notion of a kernel is a group of basic data structures and procedures that can be used to produce larger and more complex pieces of software. The Hydra operating system [WLH81] is an example of building an operating system from a set of kernel primitives that were validated and thoroughly tested. The details for a safety executive kernel would be different but many principles are applicable. The major functions of the executive kernel would be fault-tolerance, safety monitoring, and fail-safe operation.

In providing for fault-tolerance, the executive would employ a task list that specifies the execution sequence of procedures. Along with the execution sequence, the task list would also contain information about how the procedures are to be executed (e.g. parallel or switching redundancy). Each procedure would be written in a normal manner but would include assertions and other internal checks in order to return information on the routine's success to the executive.

Once a procedure has returned its status, the executive would be responsible for determining the appropriate response. If there were no problems, the execution sequence in the task list could be continued. Other possible actions would be to rearrange the order of tasks (make a faulty primary procedure the alternate in switching redundancy) or to disable a procedure that has taken too much time).

Using a task list for program structuring also enhances the ability to tune a program. Changes in where fault-tolerance is required or in how it is used would merely require changes in some tables within the executive program. The individual modules would not need to be altered, thus limiting the number of possible errors introduced while maximizing the possibility of making changes after the system is operational. For example, during testing or operation, it may become apparent that a critical module is less reliable than was originally estimated. New fault-tolerant or fail-safe procedure can then easily be implemented without major recoding efforts.

In parallel with the execution of the main procedure sequence, several safety monitoring processes can also be executing. These processes monitor critical parameters to ensure that an unsafe state is discovered and/or that an unsafe state does not lead to a safety failure. The type of monitoring done would depend on the parameter, but these

monitoring processes would communicate with the executive by interrupts to warn of unsafe situations. The executive would handle the interrupts and modify the task list to continue safe operation.

By using this interrupt driven structure, the executive would be responsible for initiating any fail-safe procedures. Thus the logic and responsibility for fail-safety is centralized and can be given top priority. The fail-safe code is separated from the main procedures and this increases the maintainability of the fail-safe procedures.

An experimental system exploring the above ideas will be developed. This system would help determine the practicality of the ideas and the merits in comparison to the other forms of software redundancy.

5.0 RECOMMENDATIONS AND GUIDELINES

The following is a summary of a few recommended techniques for designing safe software. All of them must be carefully considered when designing a system that is safety critical.

General software engineering practices, such as structured design and modular programming, should be used.

Faults found through testing or methods such as fault tree analysis should be eliminated if possible.

If faults cannot be eliminated, then measures should be taken to limit the level of the hazard. This can be done by monitoring safety critical parameters and, when an unsafe state is detected, initiating recovery activities or issuing warnings.

Start out in a safe state and only enter an unsafe state (e.g. arming a missile) when absolutely necessary.

Use capabilities to limit access to safety critical functions.

Specify a safety sequence so that any safety critical operation must be preceded by several prerequisite actions to avoid accidental activation.

Use operating systems techniques (e.g. semaphores, monitors, kernels) to avoid problems of concurrency and synchronization.

When there is still a possibility of failure use fail-safe design.

1. Fail-passive design reduces a system to its lowest energy level in the event of a failure (e.g. disarming a missile after a certain period of time).

2. Fail-active (fail-soft) design should be used in systems when the system activity must be maintained to some degree to remain safe. Malfunctioning components may be terminated (reconfigurations).

3. Fail-operational design (fault-tolerance) is used to ensure full system functionality in the event of a failure. Use redundancy (parallel or switching) for critical modules that are required to maintain safety.

Parallel (N-version programming) or switching (recovery blocks) redundancy may be more appropriate depending on the application. A combination of both techniques (using a safety executive kernel) may be useful.

6.0 PROGRAMMING LANGUAGE CONSIDERATIONS

There have been claims that Ada is not a reliable language and should not be used in safety critical applications [HOA81]. To evaluate this claim, it is necessary to understand how a programming language affects the reliability of the resulting system.

Faults enter a program by: 1) incomplete or ambiguous specifications, 2) program designers overlooking interactions among various parts of the system, 3) programmers misinterpreting the specifications, design algorithms, or data structures, 4) programmers writing a program which does not do what the programmer thinks it will do, 5) programmers misunderstanding some aspects of their programming language, or 6) mechanical errors occurring during coding, transcription, or entry. Of these, only the fourth and fifth involve features of the programming language itself. The others can be handled through management and development tools which are independent of the particular programming language being used.

Although it is not possible to prove that particular programming language features will enhance the reliability of the programming process, some empirical evidence has been collected and speculation abounds. Actual experiments have looked at particular language features, e.g. semicolon as separator versus terminator or typed versus typeless languages (for a summary see HOR79). We are unaware of any experiments which have compared "small" versus "large" languages, e.g. Fortran versus PL/1.

There has been speculation, however, that certain language attributes will enhance reliability. These include masterability, fault-proneness, understandability,

maintainability, and error-checking [HOR79]. It seems reasonable that the programmer should have mastered the language constructs and know how to use them effectively. The language should not be so complex that the programmers do not understand it in its entirety. This has been the major objection to Ada. But simplicity is not the only or even a sufficient language feature for reliability. Even more important is that the language must encourage the production of simple programs. The most powerful weapon against erroneous programs is readability. Ada, with its powerful abstraction, information hiding, and modularization facilities, should allow and even promote readable programs. Ada further encourages, and even demands, a "constructive" approach to programming and directly supports software development [BN81]. These features should also enhance the maintainability of Ada programs.

Thus although Ada can be criticized as being too large a language, there are other features of Ada which will enhance program reliability and safety. Further, it is possible to program in Ada without using all the features available. Although subsets are discouraged in order to enhance portability, the programmers need not use all the features that the compiler can handle.

7.0 CONCLUSION

The application of system safety techniques to software safety has been explored, and some resulting guidelines and recommendations for software design in light of this discussion presented. More research has to be done on the cost analysis, tradeoffs, and reliability and/or safety of these methods in practice. Some general conclusions can, however, be made.

Most software development methodologies in existence today increase reliability and hence presumably will enhance software safety. Software is certainly safer if is contains fewer errors, which is the goal of most software development procedures. However, some new techniques, or perhaps new emphasis on old techniques, are necessary to include in these methodologies. For example, critical real-time software is more similar to operating systems than to application programs, and the methodologies must be adjusted accordingly. More emphasis on potential errors is needed in the early requirements and design stages. Design techniques are needed which emphasize fail-safe procedures, provide redundancy at safety critical points, and allow the flexibility to provide different types of fault-tolerance depending on what is most appropriate for the particular function being handled. The particular technique for safe design proposed in this paper is currently under study and

development, and further results should be available shortly.

In conclusion, with regard to programming languages, Ada seems to have some features which will enhance software safety and some which will not. Although it is an interesting exercise to criticize Ada and to argue about the ways it could be improved, the important question to the potential user of the language is whether any better alternatives exist. In the opinion of the authors, they do not. No perfect language has been designed which will guarantee that programming errors will not be made. In comparison with other available languages, Ada comes out well. Further, the arguments against the use of Ada appear particularly ludicrous when one considers that much of the software for critical real-time software systems is now being written in assembly language which is by most measures the least safe language to use.

REFERENCES

[AVI75] Avizienis, A. "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing", Proceedings of the Int. Conference on Reliable Software, SIGPLAN Notices, vol. 10, no. 6, 1975, pp. 458-464.

[BN81] Brender, R. F. and I. R. Nassi. "What is Ada?", Computer, vol. 14, no. 6, June 1981, pp. 17-24.

[CA78] Chen, L. and A. Avizienis. "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Eighth Int. Conf. on Fault-Tolerant Computing, Toulouse, France, June 1978, pp. 3-9.

[CB71] Carter, W. C. , and W. G. Bouricius. "Computer Architecture and its Evaluation," Computer, vol. 4, no. 1, Jan.-Feb. 1971, pp. 9-16 .

[DIJ68] Dijkstra, E. W. "Cooperating Sequential Processes," in F. Genuys (ed.) Programming Languages, Academic Press, 1968, pp. 43-112.

[DVH66] Dennis, J. B. and E. C. Van Horn. "Programming semantics for multiprogrammed computations," CACM, vol. 3, no. 4, Mar. 1966, pp. 143-155.

[ERI81] Ericson, C. A. ,II. "Software and System Safety," Proceedings of the Fifth International System Safety Conference, Denver, CO , System Saftety Society, Inc., 1981, pp. III-B-1 - III-B-11.

[GRI81] Griggs, J. G. ,II. "A Method of Software Safety Analysis," Proceedings of the Fifth International System Safety Conference, Denver, CO , System Saftety Society, Inc., 1981, pp. III-D-1 - III-D-18.

[HAM72] Hammer, W. Handbook of System and Product Safety,Prentice-Hall, Inc. , 1972.

[HOA74] Hoare, C. A. R. "Monitors: An Operating System Structuring Concept," CACM, vol. 17, Oct. 1974, pp. 549-557.

[HOA81] Hoare, C. A. R. "The Emperor's Old Clothes," CACM, vol. 24, no. 2, Feb. 1981, pp. 75-83.

[HOR79] Horning, J. J. "Programming Languages," in T. Anderson and B. Randell (eds.) Computing Systems Reliability, Cambridge University Press, 1979.

[IEE79] IEEE Computer Society Standards Committee. Computer Dictionary (Draft Standard), Martin H. Weik (ed.), IEEE Computer Society, 1979.

[LIP79] Lipow, M. "Prediction of Software Errors," Journal of Systems and Software, vol. 1, 1979, pp. 71-75.

[RAN75] Randell, B. "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, vol. 1, no. 2, 1975, pp. 220-232.

[ROU81] Rouse W. B. "Human-Computer Interaction in the Control of Dynamic Systems ," ACM Computing Surveys, vol. 13, no. 1, Mar. 1981, pp. 71-99.

[SHA74] Shaw, A. C., The Logical Design of Operating Systems, Prentice-Hall, Inc. , 1974.

[WLH81] Wulf, W. A., R. Levin, and S. P. Harbison. HYDRA/C.mmp An Experimental Computer System, McGraw-Hill, Inc., 1981.

[WUL75] Wulf, W. A., "Reliable Hardware/Software Architecture," IEEE Transactions on Software Engineering, vol. 1, no. 2, June 1975, pp. 233-240.