N89-21754

1988

NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA

SOFTWARE SYSTEM SAFETY

Prepared by:	James G. Uber
Academic Rank:	Assistant Professor
University and Department:	The University of Alabama in Huntsville; Department of Mechanical Engineering
NASA/MSFC:	
Laboratory:	Safety, Reliability, Maintainability and Quality Assurance
Division:	Systems Safety Engineering
Branch:	Hazard Analysis
MSFC Colleague:	Dewey B. Channell
Date:	August 26, 1988
Contract No.:	NGT 01-002-099 The University of Alabama

SOFTWARE SYSTEM SAFETY

by

James G. Uber
Assistant Professor of Civil Engineering
The University of Alabama in Huntsville
Huntsville, Alabama

Abstract

Software itself is not hazardous, but since software and hardware share common interfaces there is an opportunity for software to create hazards. Further, these software systems are complex, and proven methods for the design, analysis, and measurement of software safety are not yet available. This report reviews some past software failures, future NASA software trends, software engineering methods, and tools and techniques for various software safety analyses. Recommendations to NASA SRM&QA are made based on this review.

ACKNOWLEDGEMENTS

I wish to thank the many engineers in the SRM&QA Systems Safety Division who made my stay enjoyable and who shared their thoughts and concerns about software and systems safety. Also, it was Mike Freeman and Gerald Karr who introduced me to members of the SRM&QA organization and thus made this work possible. The financial assistance provided by the NASA/ASEE summer faculty fellowship program is acknowledged.

1. INTRODUCTION

Complex software systems are often required for real-time control and monitoring of aerospace systems hardware. The NASA Space Shuttle software, for example, consists of some ten million lines of computer code that control and monitor hardware on the ground and in the air; the Shuttle could not fly without it. Since software and hardware share common interfaces, there are opportunities for software to create hazards. This linkage has historically been ignored, however, in formal analyses of system safety.

This report examines some approaches for identifying and eliminating or controlling software hazards. In order to appreciate the software safety problem, however, it is necessary to understand something about software development practices, the characteristics of software and software errors, and "the reasons why things are the way they are." Accordingly, much of the report text is devoted to these preliminary issues in preparation for a later discussion of software safety and software safety analysis methods.

Section three provides the motivation for continuing on into the rest of the report. This motivation is provided by discussions of software development "horror stories," future trends in NASA software control systems development, and the value of implementing software safety programs. The next section discusses the current state of software engineering, some reasons for the current situation, and some modern approaches to software development. Section five considers the software safety problem and some solutions, and section six presents recommendations for addressing present and future software safety problems.

2. Objectives

The objectives of this study are to explore the breadth and depth of the software system safety problem, and to recommend SRM&QA actions that can reduce the present and future hazards posed by software.

3. MOTIVATION

3.1. Statistics and Failures

Does the Federal Government get what it pays for? Perhaps not, at least when it comes to software development. A decade-old study of the software costs associated with nine federal projects depicts software procurement as a monetary black hole [Neumann, 1985]. Of software expenditures totalling $6.8 million, 95% ($6.5 million) was wasted on software that either was delivered but never used (47%), paid for but never delivered (29%), or abandoned or reworked (19%). Less than 2% of the software expenditures yielded software that was used as delivered. These statistics reflect the inherent difficulty of large-scale software management and development, although improvements in software engineering practice may produce more impressive results if a similar study were conducted today.

Problems with real-time control software may be subtle and remain innocuous until the worst possible moment. These problems can also survive supposedly rigorous testing procedures. A good example of such a software problem is the "bug" that delayed the first shuttle orbital flight on April 10, 1981. The bug was discovered when, twenty minutes prior to scheduled launch, the backup flight control system (BFS) failed to initialize when commanded. Curiously, it turned out that the bug was not in the BFS, but was in the primary avionics software system (PASS), and had gone undetected for over a year. John Garman, who at the time was Deputy Chief of the Spacecraft Software Division at JSC, gives a detailed account of the discovery and correction of the bug [Garman, 1981]; some of the details of this very intricate and subtle problem are given below.

There are five general purpose computers (GPC's) onboard the Shuttle. These five independent machines combined with the PASS and BFS provide two different types of redundancy to guard against two different types of failures. The PASS executes on four of the GPC's and provides for full operational capability after the failure of one GPC and a safe return capability after the failure of a second. Note, however, that this redundancy provides protection only against hardware failures. Potential hazards from software design and implementation errors are not controlled since the identical PASS executes on all four machines simultaneously; a single software error (i.e. a common mode failure) could "crash" all four GPC's and, in Garman's words, convert "... the Orbiter to an inert mass of tiles, wires, and airframe..." From this

realization came the idea to include a fifth GPC, which would execute a functionally equivalent but separately designed and implemented flight control system, the BFS. It was (or is) believed that this "software redundancy" would contribute to the overall safety and reliability of the flight software system.[1]

In its standby mode, the BFS operates by "listening" to the sensor inputs and some of the outputs from the PASS. Thus the BFS is constantly aware of the current state of the Orbiter and ready to "take over" when necessary (a crew decision). The BFS must stop listening, however, if it detects an inconsistency in the PASS's data processing, so that the integrity of the BFS can not be compromised by a failure in the PASS (a separation of failure modes). On the day of the bug incident, the BFS was tricked into believing that the PASS was compromising its data, and so properly refused to initialize (i.e. start listening to the PASS). The BFS was tricked because the BFS processes (processes can be thought of as individual tasks or programs) and some, but not all, of the PASS processes were out of phase with other PASS processes that were responsible for the "polling" of data from sensors. Thus the BFS regarded all the information from the sensors as garbage. This out-of-phase condition occured because the BFS processes and some, but not all, of the PASS processes were scheduled based on a cycle counter, while the remaining PASS processes were scheduled based on a separate calculated startup time.[2] The problem was that this calculated startup time appeared, to the computer, to be in the past because it was compared to a value in a supposedly empty timer queue that unfortunately was not empty. The computer let the "past" start time "slip" into the future (as would an alarm clock set to go off an hour ago) and hence the out-of-phase condition. The queue was not really empty

---

[1] This is debatable, even among the system developers. The BFS adds significant complexity to the system (as will be discussed), and in fact the present failure was an indirect result of the BFS implementation. Further, this "software redundancy" concept is essentially like the "N-version" approach to achieving software reliability, which has its detractors. Both rely on the assumption that software design and implementation errors for functionally equivalent but independently developed systems are in fact independent; this assumption may not be justified, as discussed in a later section. In any case, the debate should not center on whether or not the development of the BFS system improved reliability and safety. The debate should instead center on whether the costs of BFS development would have been better spent elsewhere, e.g. on further design and testing of the PASS. One wonders to what extent development of the BFS was motivated by uncertainty about the trustworthiness of the PASS, or by an ability to "point the finger" in the case of a disaster involving flight software. As Garman says, "(because of the presence of the BFS) almost everyone involved in the PASS-side 'feels' a lot more comfortable!"

[2] If you find this incredibly complex and confusing, don't worry - it is. The point here is not to try to understand all the detail, but rather to appreciate the subtlety and complexity of problems that can and do occur in real-time control system software. To me, the PASS/BFS design sounds like an SRM&QA nightmare.

because an isolated data initialization process put a "delay" value in the timer queue that specified when to start the initialization routine. This delay was initially small and did not produce any unwanted side effects. But late in the system development, about one year prior to launch, this isolated value, a single "constant in the code," was increased to a value large enough so that the out-of-phase condition was possible (the value was increased because of a totally unrelated problem, and by this time much of the system testing and integration had been completed). It is significant that much of the problem stemed from a basic incompatibility between the BFS and the PASS; the PASS is an asynchronous system (i.e. a priority interrupt system) while the BFS is a synchronous system. The PASS system had to be made to "look synchronous" to the BFS, but the emulated synchronism was implemented unevenly (only in "critical" processes), and so the possibility arose for processes to become out-of-phase.

The bug that delayed the initial orbital launch of the Space Shuttle is one particularly well documented example of the types of problems that can occur in real-time software.[3] As shown, problems can be caused by seemingly inconsequential changes to seemingly unrelated code segments. Note also that the timing of events is important, yet the timing constraints that must be satisfied for safe operation may be unknown or unclear. The following statement by Garman is appropriate:

> "It is complexity of design and process that got us (and Murphy's Law!). Complexity in the sense that we, the 'software industry' are still naive and forge into large systems such as this (the Shuttle) with too little computer, budget, schedule, and definition of the software role. We do it because these systems won't work, can't work, without computers and software."

Garman also notes that the ability to quickly and easily modify software (unlike hardware) is a two-edged sword; this flexibility can reduce costs and lead to better designs, but uncontrolled flexibility can create disasters. Recently, a Shuttle engine test was scrubbed by a computer software check because of an apparently sluggish valve. The valve controls the flow of liquid hydrogen and must be no more than 20% open to prevent a fuel system rupture. According to an Associated Press report in The Huntsville Times (August 5, 1988), NASA had the options of replacing the valve or changing software

[3] Many other illustrative examples have been recorded over the past decade in the issues of Software Engineering Notes [Neumann, 1985].

commands to accomodate the way the valve worked (?) during the test. NASA officials decided to change the valve.

3.2. Future Trends

Software controls have many advantages over hardware controls, including greater precision, flexibility, and (perhaps) reliability, as well as allowing a high degree of automation. So software will continue to be an important part of future aerospace systems, and more of the same types of problems can be expected. The Space Station, for example, will depend on the Space Station Information System (SSIS), Data Management System (DMS), Technical and Management Information System (TMIS), and other communications, tracking, and ground support software to perform its important information gathering functions. Further, these software elements will need to interface successfully with each other and with crew members, laboratory experiments, and (life critical) software controlled subsystems such as the Environmental Control and Life Support System (ECLSS).

Perhaps more significant for the future are the new set of software safety problems posed by the incorporation of "advanced software technology" into the Space Station operations software. The use of artificial intelligence and expert system technology is mandated by the Space Station Program office. Consider the following Space Station requirements [NASA, 1988a]:

> "(capability for) Growth of artificial intelligence and robotic technology"
>
> "Incorporation of machine intelligence in the form of expert systems, initially for well-defined and structured applications and later for more advanced applications"
>
> "Complex information interfaces of telerobots and autonomous robots. The following are example types of potential information required by the intelligence of robotic devices:
>
> --The location at all times
> --The location of obstacles and how to avoid them at all times
> --The proper interaction sequence with objects to be manipulated
> --Status of itself and its task object
> --Its own limitations with respect to its current environment."

These advanced technology requirements either require new software concepts (e.g. artificial intelligence and expert systems) or require new software applications (e.g. control of telerobots and autonomous robots in hazardous space environments). These requirements add significant complexity to the envisioned Space Station software system and add new types of safety and reliability problems that are not found in other real-time control and monitoring systems. Unfortunately, common software engineering management and development practices may be ineffective against these new problems, as they have been against the more traditional problems of the past.

3.3. Failure Costs

Some would argue that serious measures to improve software safety in the manned space program are unjustified because of added costs and limited funds. The space program is, after all, a large-scale research program, so why doesn't NASA just "get on with it," and put the hardware into space without worrying so much about the safety of this or the reliability of that. This seems to be the view of Senator Jake Garn, R-Utah, who flew on Discovery in April 1985. In an August 5, 1988 Associated Press article in The Huntsville Times, Garn was quoted as saying "I think that at this point we probably are being a little bit too cautious because of all the attention (from the news media)." Garn may be right. After all, the potential catastrophic impact from a manned space disaster surely cannot compare with, for example, the accidental firing of a nuclear missle or the core meltdown of a nuclear reactor. This simplistic analysis is flawed, however, because it does not adequately take into account the costs of failure due to lack of appropriate safety measures.

Any rational decision on how much to spend on software safety, and systems safety in general, must consider not only the costs of implementing the safety measures but also the benefits of implementing the safety measures, i.e. the costs of disasters avoided. For the manned space program these benefits can be very great. The public no longer views NASA as simply a large-scale research and development agency. According to John E. Pike, a space program analyst at the American Federation of Scientists, "A lot of the public support and interest (in NASA programs) grows out of the perception that this represents the best of America and our highest aspirations" (The Huntsville Times, August 7, 1988). As a consequence, NASA's public image can suffer greatly because of failures. In a recent survey of 1,223 adults, nearly half lost confidence in NASA after the Challenger disaster, and 60 percent of those still lacked confidence (The Huntsville Times, July 25, 1988). The effect of future

failures on agency funding should be obvious. According to Pike, "If they (NASA) don't get another dozen flights under their belt before another accident, I'd be real surprised if the shuttle ever flew again. If (an accident strikes) this one (the next Shuttle launch), it would pretty well stop the space program." Such high costs of failure suggest that safety measures, including software safety measures, are probably a bargain if they can measureably reduce the chances of failure.

4. SOFTWARE ENGINEERING

4.1. Current State

Software engineering is the art or science or whatever of designing, constructing, and testing computer programs. David Parnas, a noted computer scientist and consultant, wrote a series of eight essays on why the Strategic Defense Initiative would not be trustworthy because of software development difficulties [Parnas, 1985]. These essays were submitted in 1985 along with Parnas' resignation from the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization. In one of these essays Parnas suggests that the state-of-the-art in software engineering is significantly behind that of other more mature engineering disciplines. His claim is supported by three interesting contrasts between software and other engineered products:

1) When most engineered products are designed, tested, and sold, it is assumed that the product is "correct" as per functional requirements) and reliable. Finished software, on the other hand, often has significant "bugs" and may be unreliable.[4] It is usually expected that the software will improve with subsequent versions, but this is not always the case.

2) Most engineered products come with an express or implied warranty, but finished software products often come with a specific disclaimer of warranty.

3) Designers in more traditional engineering disciplines have been educated to understand and use a variety of mathematical tools, while designers of software are generally uneducated in even the modest tools that are available to software engineers.

Some reasons for the current state of software engineering are given below. The first reason is suggested by Parnas, the others are my personal observations.

---

[4] Respected computer scientist Edsger W. Dijkstra [Neumann, 1985], claims that "most of NASA's software is full of bugs." After the successful moon landing in 1969, Dijkstra asked Joel Aron of IBM, who had been responsible for much of the flight software development, how he "got that software to work okay." "Okay?" Aron replied, "It was full of bugs. In one of the trajectory computations, we had the moon's gravity repulsive rather than attractive, and this was discovered by accident five days before count zero."

Software is complex.

One reason for the current state of software engineering is the inherent complexity of large-scale software projects. Large software systems are discrete systems with an enormous number of possible states. Further, software systems usually have few repeatable structures, so that it is not possible to construct a large software system by assembling a large number of small, identical, software modules.[5] In contrast the design of large-scale integrated circuits, while discrete and having many possible states, is made simpler because many of the structures are repeatable (note that modern computer hardware is vastly more reliable than computer software).

Success in software development may, in fact, be limited by the complexity of the underlying system. In the conclusion to one of his essays, titled The limits of software engineering methods, Parnas discusses the difficulty of developing a trustworthy software system for SDI battle management:

> "I am not a modest man. I believe that I have as sound and broad an understanding of the problems of software engineering as anyone that I know. If you gave me the job of building the system (SDI battle management software), and all the resources that I wanted, I could not do it. I don't expect the next 20 years of research to change that fact."

Software is abstract.

Software only exists in the computer, which is very different from the world that people live in. Software can not be seen, touched, or heard, it does not have material strength, and it does not bend, twist, chip, split, or otherwise wear out. This abstract property makes software design appear inefficient to the near-sighted, because design and coding (i.e. "construction") cost about the same but coding produces something tangible (the code can be run on a computer and large piles of results can be printed). Thus there is a strong tendency in software engineering to skip over the design phase to the coding phase. In many other engineering disciplines, however, there is a clear cost advantage to establishing rational and efficient design

---

[5] Neumann [1984] quotes from a book called Software and Its Development by Joseph Fox, where Fox discusses the problems of testing Air Traffic Control software systems: "The number of possible paths through these large programs, and the number of possible combinations of states of inputs, data, calculations, and interactions is so large that even in 100 years of use, we will only be beginning to execute the first few percent of the possible paths (through the code). Even after years of real use, there will still be bugs in the program."

methods because an iterative "construct and test" approach is too expensive.

Software design requirements may be ambiguous or incorrect.

This may be just a result of the complexity of software as discussed by Parnas, but it is important enough to deserve separate mention. Software requirements are often written without specific knowledge of the hardware and without complete knowledge of the functional requirements. In contrast a civil engineer, for example, can make logical assumptions about the loads a structure must withstand and the "worst case" combination of loads.

Software engineering is multidisciplinary.

It seems intuitive that a software designer should understand the application area as well as the software engineering discipline (remember that computer science is fundamentally an applied discipline). Unfortunately, these larger-than-life persons are difficult to locate, and in any case would probably demand far too much money. As a result, software is usually written by persons who only know software or who only know the application.[6]

4.2. Modern Approaches

The current state of softwear engineering should concern those responsible for the safety, reliability, and quality of softwear. In recent years several softwear engineering design approaches and tools have been developed to aid the softwear engineering manager and the softwear designer. These approaches and tools include: the concept of the softwear acquisition life cycle; independent verification and validation of softwear projects; and computer-aided softwear engineering tools. Although these concepts and methods are not a panacea-they do not guarantee high quality softwear-they can, if implimented uniformly, be the foundation for other design approaches and techniques aimed at, for example, improving softwear safety. Uniform implimentation of these ideas also allows accurate measurement of the effects of future changes in softwear design approaches. Each of these design approaches and tools is discussed in the following paragraphs.

---

[6] I read a recent letter to the editor that argued for a return to the "good old days" when computer scientists were fundamentally rooted in an area of application (e.g. some type of engineering) and applied computer science principles to their field of interest. This person suggested that the four year undergraduate degree in computer science be eliminated in favor of a two year graduate degree.

The software acquisition life cycle (SALC).

The SALC comprises a heirarchical set of software design, coding, and maintenance tasks, and associated documentation, that define the software acquisition process from initial concept phases through final delivery and maintenance phases. The SALC is a highly structured approach to softwear engineering. Specific requirements associated with each phase are satisfied and baselined before the next phase is begun. Control is exercised at the transitions between phases to assure consistency of requirements and specifications.

The NASA SALC is shown in Figure 1, which is taken from the NASA guidebook titled: Software Verification and Validation for Project Managers [NASA, 1987a].[7] Loesh [1988] gives a general description of the objectives of the various life cycle phases, which for convenience is repeated below.

1) Concept and Project Definition Phase. Ensure software plans, policies and management are appropriately part of early plans and design, i.e. feasible/scoped/costed.

2) Software Initiation Phase. Specify a formal system/software definition and consummate the acquisition/development plans (lowest risk delivery agreements).

3) Software Requirements Definition Phase. Scope each program, establish engineering procedures and confirm computer/software compatibility (last chance to say what you want).

4) Software Preliminary Design Phase. Evaluate the following: 1) are there any major design flaws, and 2) is it consistent with the hardware it is planned to execute to?

5) Software Detailed Design Phase. Determine that the module does the right things and will be coded following good engineering principles.

6) Software Implementation Phase. Build and show as agreed that the program works correctly.

7) Software and System Integration and Testing Phase. Does the system do what was requested?

[7] This guidebook has been updated recently, and the SALC in the new version may be somewhat different.

ORIGINAL PAGE IS
OF POOR QUALITY

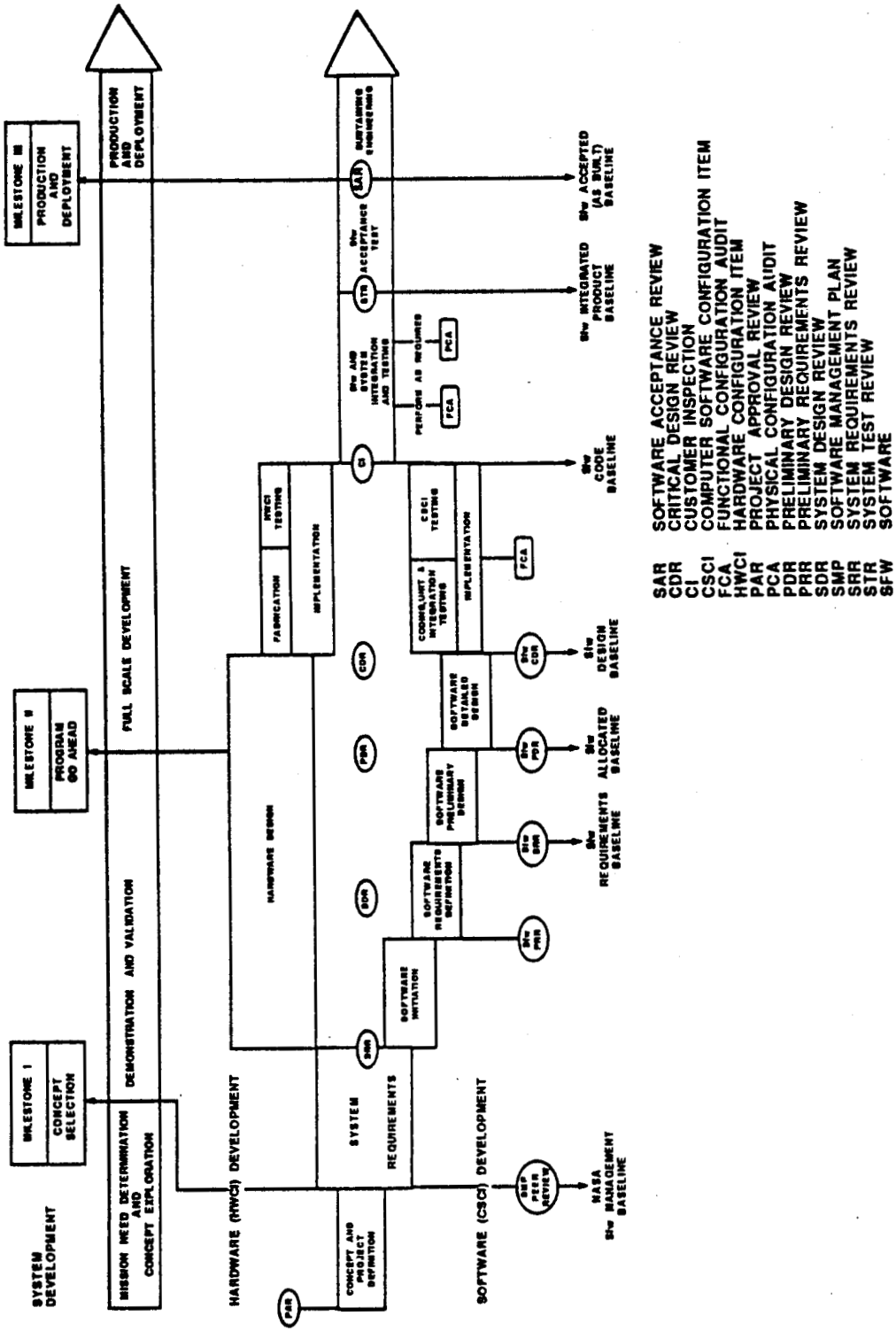

Figure 1 - Software Acquisition Life Cycle

8) Software Acceptance Testing Phase. Get the software and pay the bill.

9) Sustaining Engineering Phase. Fix latent softwear bugs and produce product upgrades, while ensuring that the software still works correctly. Manage software distribution and archival (including documentation).

It should be obvious that successful implementation of the SALC concept does not come easily and without costs. It has been demonstrated in practice, however, that the life cycle concept is cost efficient when considering costs over the entire useful life of the software (recall the statistics on military software spending). The life cycle model is also thought to improve the general quality of the finished software product (which of course can be linked to cost in terms of sales revenues, disasters avoided, and reduced maintenance). Perhaps the real reason why the SALC works is that actual software coding (implementation phase) is postponed till late in the project. The software managers and designers must think hard in the beginning about requirements, specifications, and associated design issues. This process (and the sheer volume of associated paperwork) allows the designers and implementers enough time to mull over the various design possibilities and choose a good one before committing to code, all the while "appearing" to be productive. Some successful software engineering firms have implemented rigid policies so that on a one-year project, for instance, coding cannot commence until the ninth month! Another important function of the SALC is that it provides the development framework necessary for implementing design tools and procedures aimed at accomplishing more specific software design objectives, e.g. increasing software safety.

Independent verification and validation (IV&V).

Software independent verification and validation is a technical discipline that is in effect during all phases of the software acquisition life cycle. The various IV&V activities and their relations to the SALC phases are shown in Figure 2, taken from the previously cited NASA guidebook. As shown, IV&V activities include designing and executing tests based on information from requirements and specification documents, design documents, and code audits. These tests are meant to ensure that the final delivered product meets the stated requirements and is of high quality. In addition to formal testing, IV&V activities also assure that each software end item, whether a individual module, a partially integrated subsystem, or a fully integrated system, satisfies its corresponding requirements and specifications as prescribed during the heirarchical design process (a definition of

ORIGINAL PAGE IS
OF POOR QUALITY

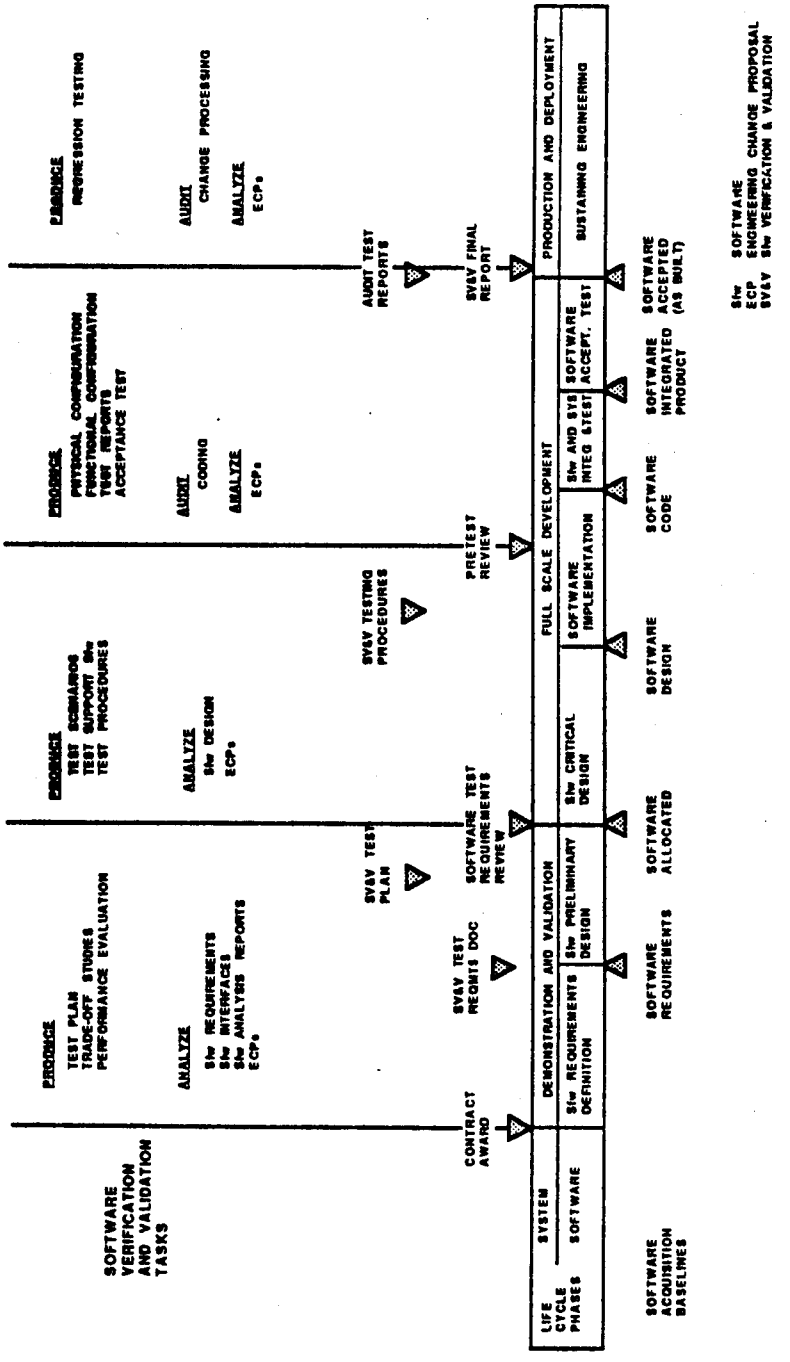

Figure 2 - Software IV&V Activities

"validation"), and that the products of each phase of the SALC satisfy the requirements and specifications of the preceeding phase (a definition of "verification"). It seems logical that IV&V staff should be mostly seasoned, experienced personnel (who knows all the "tricks of the trade"); this is no place for the green programmer with little practical experience.

The "independent" in IV&V refers to the relationship between the project management (NASA), the development contractor, and the IV&V contractor. The quality of the IV&V effort and hence the quality of the delivered softwear is thought to increase with the degree of independence between these players. Minimal independence is achieved by a development contractor who uses the development staff for IV&V (really just V&V), while maximal independence is achieved by a wholly separate IV&V contractor that communicates with the developer and with NASA by separate channels. A truly independent verification and validation contractor provides an unbiased critical review and "second opinion" of the softwear. Independence assures that the testing and design reviews will not be polluted by the day-to-day design activities, and that IV&V personnel will not be competing for resources with development personnel. A higher degree of independence does, of course, add significantly to the up-front project cost. These costs should properly be weighed against the criticality of the software when deciding on the degree of independence. This trade-off analysis should also consider the non-trivial costs of the sustaining engineering phase (maintenance), which are presumed to decrease with an increase in independence.

Computer aided software engineering (CASE).

CASE tools are programs to help write programs. CASE tools emphasize a systems approach to software engineering, as opposed to a ad hoc "software crafting" approach. Thus CASE supports the life cycle model of software development, and can make the transition to SALC easier by assisting the programmer in various tasks associated with the life cycle phases. The following are examples of CASE tools that either exist or are under development: program design languages and specification languages; test design aids; symbolic debuggers; automatic code generators; and configuration management aids. These tools are usually graphical and highly interactive. Other CASE tools implement the structured approach to requirements specification that is growing in popularity [DeMarco, 1979].

The future Space Station Software Support Environment (SSE) is an ambitious effort by NASA to supply Space Station software developers with a consistent and comprehensive set

of CASE tools. Consider just a few of the stated requirements for the SSE, taken from the SSE system functional requirements specification [NASA, 1988b]:

> "The SSE system shall support project initiation and control, project management, metrics collection and analysis, project planning, budgeting, resource allocation and accounting, scheduling, lessons learned gathering and analysis, and performance management."
>
> "The SSE system shall provide the capability to determine the complexity and criticality of a proposed change and to prioritize proposed changes based upon those metrics."
>
> "The SSE system shall support computer-aided preliminary design for SSP operational software and SSP prototypes, models and simulations."
>
> "The SSE system shall support software checkout and verification from the unit level, to the subsystem level, to the complete software system, and finally to integrated software and hardware systems."

And the list of requirements goes on and on. Once complete, the SSE will be a state-of-the-art programming environment for the development of all Space Station software. Further, the SSE will support the development of new software engineering tools and methods as well as their rapid integration into the development life cycle. Some of these new tools and methods could address timely issues in software safety.

5. SOFTWARE SAFETY

Software by itself is harmless. When software is embedded in a hardware system, however, say for a real-time control application, inappropriate software actions or inactions can lead to hazardous system states with catastrophic potential. Nevertheless, modern software development efforts do not normally consider these software safety concerns. Safety is not usually included in the software requirements specification, and IV&V activities address software "quality" issues but not necessarily software safety.

Comprehensive and generally applicable software safety methods, tools, and techniques do not currently exist. Software safety can be achieved only through software design (as opposed to redundancy as in hardware). Unfortunately, there are few adequate mathematical tools for analyzing software designs, and few adequate metrics for measuring software safety. It is unlikely that a "safety factor" concept can be applied to software, and at the present there is no comprehensive "building code" that will assure software safety.

Although technology lags behind, there is a strong institutional motivation for developing and implementing software safety methods. Applicable government standards and new project requirements are now addressing software safety issues. This new focus is especially noticeable in the Military safety program requirements and the program requirements for the Space Station.

The Military standard on system safety program requirements [DoD, 1987] includes a new task section (as of July 1987) on software hazard analysis requirements. This set of requirements emphasizes a systems approach (which is necessary, of course) that includes analyses of the safety critical hardware/software interfaces. Thus information from hardware safety analyses (hazard analyses, FMEA's) are used as input to the software safety analysis, and leads to the concept of a Safety Critical Computer Software Component (SCCSC). These SCCSC's or sets of interfacing SCCSC's can be targeted for detailed investigations to determine, for example, whether certain hazardous events can occur as a result of software actions or inactions, or if adequate software safety controls are present. This standard appropriately supports the life cycle software development model, and includes software hazard analysis requirments for each of the major SALC phases.

A variety of Space Station Program (SSP) documents specify software safety requirements. The software product assurance requirements [NASA, 1988a] state that the provider "shall plan, document, and implement a software safety process." The MSFC level C Space Station software management plan [NASA, 1987b] requires a software product assurance plan that emphasizes "software fault tolerance and software failure mode and effects analysis for all critical software packages." The instructions for SSP hazard analysis preparation [NASA, 1987c] includes software hazard analyses in the scope of work, and describes a software hazard analysis approach that includes the use of software fault tree analysis.

The cited Military standards and NASA documents say what to do but not how to do it (which is appropriate for documents of this type). As indicated above, the current technology for the "how" part is probably inadequate for implementing the "what" part. Recent concern about software safety has, however, spurred the investigation of new software safety tools, techniques, and methods [Leveson, 1986]; these approaches are summarized below. Most investigations have borrowed existing hardware safety analysis techniques and applied them to software, and most have applied the techniques to relatively simple test systems. Although in an early stage of development, some of the techniques show promise, and the continued development of these and other software safety approaches should be supported in order to meet the software safety needs of the future.

Software fault tree analysis (SFTA).

The application of fault tree analysis to software was investigated by Leveson and Harvey [1983], Leveson and Stolzy [1983], Leveson [1984], and Cha, Leveson, and Shimeall [1987]. SFTA uses a knowledge of the programming language to build a fault tree that describes all possible software paths leading to a particular hazardous event. Successful paths (those which produce the hazardous event) must be eliminated or controlled or justified as very unlikely to occur in practice. In Leveson and Harvey [1983], SFTA was applied successfully to a flight and telemetry control program for a University of California, Berkely, spacecraft (approximately 1250 lines of assembly language code). The SFTA identified a logic error whereby two sun-pulse interrupts within 64 ms of each other could crash the microprocessor and render the spacecraft useless (such a condition is highly unlikely, but one of the "sun-pulses" could be artificially produced by gamma radiation). A simple software check could eliminate this hazardous condition.

SFTA is practical when there exists only a limited number of safety critical failures, a condition that is apparently often satisfied. SFTA is thought to be helpful because it provides a structured approach to thinking about software problems from a safety perspective. It also may be possible to automate the generation of the software fault trees by examination of the code. A disadvantage of SFTA is the difficulty of including timing related failures in the analysis.

Petri nets.

The application of Petri nets to software safety analysis was investigated by Leveson and Stolzy [1987]. A Petri net is a dynamic system modeling approach consisting of places, transitions between places, and a set of tokens marking the places and defining the current system state. Software systems can be modeled by Petri nets and the Petri nets can be executed, which consists of "firing" the tokens along the enabled transitions toward a new set of places. Repeated firing of the Petri net models the dynamic system behavior. The advantage over static analysis approaches (e.g. SFTA) is that important timing properties of programs can be investigated. Run-time faults and failures can also be incorporated into the analysis, and there is significant opportunity for computer-aided analysis. In the paper by Leveson and Stolzy a Petri net model was applied to a simple railroad crossing controller.

Sneak software analysis.

In sneak software analysis the program is converted into flow diagrams using electrical symbols, and is then analyzed to detect certain logic errors such as undefined variables. Leveson [1986] points out that much of this information is provided by good compilers, and that furthermore sneak software analysis is more a reliability than a safety technique since it attempts to identify all faults. She further notes that it is unlikely that many significant faults will be found this way (of the type that were discussed previously), and draws a parallel between trying to find significant errors using sneak software analysis and "trying to find the errors in a book by checking the grammar."

N-version programming.

N-version programming was investigated by Avizienis [1985] as an approach to fault-tolerant computing. In this approach N different programs are developed from the same functional specifications. The developers should be as independent as possible, and should use different tools and

compilers. These N program versions are executed simultaneously and the results are polled and compared by a voting program, with the correct answer taken as the majority answer. Sophisticated methods have been devised to provide for communications between the various versions and the voting program, and hence to ease development efforts. The approach depends, however, on the assumption that software design and implementation errors associated with the different versions are independent (a high degree of dependence of errors could create situations in which the majority of program answers agree but are all wrong). Unfortunately the approach may be limited by this assumption, as a recent study suggests that the assumption of independence may not be justified [Knight and Leveson, 1986] (consider the special problem of dependence because of common specification errors). Further study is needed to evaluate thoroughly the assumption of independence before N-version programming can be used widely with confidence.

6. CONCLUSIONS AND RECOMMENDATIONS

Large-scale software systems are often used to control real-time flight and ground operations on manned space missions. This trend is expected to increase because of the emphasis on automation and advanced software technology. There is ample evidence that suggests these large software programs will contain logic errors that may under certain situations lead to hazardous conditions. Thus it is important to emphasize safety-related approaches to software development throughout the useful life of the software, beginning at requirements development. Software safety engineering is a young discipline, however, and the necessary analysis and development tools are only beginning to be developed. It is a tough problem, since software is highly complex, abstract, and does not lend itself easily to mathematical analysis.

SRM&QA should take a leadership role in the development and refinement of software engineering methods, techniques, and tools that can help to improve software safety, reliability, and quality. This is especially important for software safety, because while software reliability and quality have many proponents, software safety is largely neglected because it is separated from functional requirements and traditional performance measures. SRM&QA should strive to build and maintain a knowledge base of sound software engineering practices that it can distribute to contractors for use in project development.

Several specific recommendations follow, not necessarily in any order.

SRM&QA should continue to develop its software product assurance function and to promote SPA methods.

The SPA activities (including IV&V) are the backbone of any effort to improve software safety, reliability, or quality. SRM&QA should logically be responsible for the bulk of NASA SPA activities and in fact for the bulk of NASA IV&V contracting and monitoring. NASA should consider a software project management structure that provides for internal independence of development and IV&V engineers and managers. SRM&QA should develop a consistent and rational method for determining the necessary degree of independence of the IV&V contractor based on software criticality, total life cycle cost, and other factors.

SRM&QA should develop a software hazard analysis methodology.

This development effort may include an evaluation of applicable Military standards and other guidebooks and instruction manuals with regard to NASA's needs; an evaluation of personnel requirements including skill and education levels; identification and evaluation of existing techniques and computer-based tools for software hazard analysis; development and evaluation of new techniques and computer-based tools for software hazard analysis; and investigation and development of recommended software design methodologies that encourage the production of safe software (i.e. safety metrics and analysis techniques, and "building codes" for safe software development).

SRM&QA should explore the possibility of developing an integrated computer-based environment to support all SRM&QA management and engineering activities.

This computer-based environment would be graphics-based, would include a variety of tools specific to SRM&QA activities, and would interface directly or indirectly with the Technical and Management Information System (TMIS), the Software Support Environment (SSE), the Space Station Information System (SSIS), and the Data Management System (DMS). The potential benefits of this development effort include improved efficiency and quality of work, and expedited flow of information between SRM&QA, other NASA organizations, and contractors.

SRM&QA should develop and investigate methods, techniques, and tools for evaluating the safety, reliability, and quality of software systems that use advanced software engineering technology.

Specifically, SRM&QA should investigate special safety, reliability, and quality problems (and their possible solutions) created by the use of artificial intelligence and knowledge based expert system technology in critical software systems. Use of this technology (or provision for its use) is a requirement of the Space Station Program, and it is not clear, for example, whether current software engineering practices apply or whether anyone even knows how to debug an expert system or artificial intelligence program.

7. REFERENCES AND SELECTED BIBLIOGRAPHY

1. Avizienis, A., "The N-Version Approach to Fault-Tolerant Software," IEEE Trans. on Software Eng., Vol. SE-11, No. 12, Dec. 1985.

2. Cha, S., Leveson, N., and Shimeall, T., Safety Verification of Ada Programs in MURPHY, University of California at Irvine, UCI Technical Report 87-27, 1987.

3. DeMarco, T., Structured Analysis and System Specification, Prentice Hall, Englewood Cliffs, N.J., 1979.

4. DoD, Software System Safety Handbook, Air Force Handbook AFISC SSH 1-1, Sep. 1985.

5. DoD, Military Standard, System Safety Program Requirements, MIL-STD-882B, July 1987.

6. Garman, J., "The 'Bug' Heard 'Round the World," Software Engineering Notes, Vol. 6, No. 5, Oct. 1981.

7. IEEE, The Small Computer Revolution, Proceedings COMPCON, IEEE Computer Society Press, Fall 1984.

8. Jahanian, F., and Mok, A., "Safety Analysis of Timing Properties in Real-Time Systems," IEEE Trans. on Software Eng., Vol. SE-12, No. 9, Sep 1986.

9. Knight, J., and Leveson, N., "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," IEEE Trans. on Software Eng., Vol. SE-12, No. 1, Jan. 1986.

10. Leveson, N., "Software Safety in Computer Controlled Systems," IEEE Computer, Feb. 1984.

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

12. Leveson, N., and Harvey, P., "Analyzing Software Safety," IEEE Trans. on Software Eng., Vol. SE-9, No. 5, Sep. 1983.

13. Leveson, N., and Stolzy, J., "Safety Analysis of Ada Programs Using Fault Trees," IEEE Trans. on Reliability, Vol. R-32, No. 5, Dec. 1983.

14. Leveson, N., and Stolzy, J., "Safety Analysis Using Petri Nets," IEEE Trans. on Software Eng., Vol. SE-13, No. 3, March 1987.

15. Loesh, R., Software Verification and Validation, Training Course Notes, System Technology Institute, Inc., Rev. C6, 1988.

16. NASA, Software Verification and Validation for Project Managers, Version 0.1, Safety, Reliability, Maintainability, and Quality Assurance Publication D-GL-13, March 1987a.

17. NASA, Level C Space Station Software Management Plan, George C. Marshall Space Flight Center, SS-PLAN-0006, 1987b.

18. NASA, Instructions for Preparation of Hazard Analyses for the Space Station, JSC 30309, Space Station Program Office, 1987c.

19. NASA, Space Station Program Definition and Requirements, JSC 30000, Lyndon B. Johnson Space Center, Houston, Texas, 1988a.

20. NASA, Space Station software Support Environment System Functional Requirements Specification, LMSC F255416, 1988b.

21. Neumann, P., "Letter from the Editor," Software Engineering Notes, Vol. 10, No. 5, Oct. 1985.

22. Neumann, P., "On Hierarchical Design of Computer Systems for Critical ApplicationsIEEE Transactions on Software Eng., Vol. SE-12, No. 9, Sep. 1986.

23. Parnas, D., "Software Aspects of Strategic Defense Systems," Communications of the ACM, Vol. 28, No. 12, December, 1985.