NASA Technical Memorandum 107402, Part 1
IEEE–155NO897–5000

Software Design Improvements

Part 1: Software Benefits and Limitations

Vincent R. Lalli
Lewis Research Center
Cleveland, Ohio

Michael H. Packard
Raytheon Engineers and Constructors
Brook Park, Ohio

Tom Ziemianski
Texas Instruments Inc.
Dallas, Texas

Prepared for
The International Symposium on Product Quality and Integrity
cosponsored by AIAA, ASQC/RD, ASQC/ED, IEEE/RS, IES, IIE,
SAE, SOLE, SRE, and SSS
Philadelphia, Pennsylvania, January 13–16, 1997

National Aeronautics and
Space Administration

SOFTWARE DESIGN IMPROVEMENTS
PART I -- SOFTWARE BENEFITS AND LIMITATIONS

1. INTRODUCTION

Computer hardware and associated software have been used for many years to process accounting information, to analyze test data and to perform engineering analysis. Now computers and software also control everything from automobiles to washing machines and the number and type of applications are growing at an exponential rate. The size of individual programs has shown similar growth. Furthermore, software and hardware are used to monitor and/or control potentially dangerous products and safety-critical systems. These uses include everything from airplanes and braking systems to medical devices and nuclear plants. The question is: how can this hardware and software be made more reliable? Also, how can software quality be improved? What methodology needs to be provided on large and small software products to improve the design and how can software be verified?

Fig 1. **SOFTWARE BENEFITS**

-Reduction in weight.
-Better system optimization.
-Autonomous action can be taken by software in emergencies.
-More features are given to users of computer based products.
-System capabilities increased with computers (communication bandwidth, tuning precision, etc.).
-Better design analysis of system.
-Better knowledge of causes of system problems.

1.1. Software Reliability

Fig 2. **SOFTWARE RELIABILITY**

-Software reliability is defined as the probability that the software (actually the computer and its software) will not cause a failure of a system, or that software will not cause unanticipated conditions that could result in the loss of a system or subsystems.

Software reliability includes the probability that the program (again thinking in terms of the computer and its software) being executed will not deliver erroneous output. People have come to trust computer generated results (assuming they think the input data is correct). However, now we begin to encounter problems. Recently a manufacturer reported that its motherboards using a particular IDE (Integrated Drive Electronics) controller "when using certain operating systems have the potential for data corruption that could manifest itself as a misspelled word in a document, incorrect values or account balances in accounting software, ... or even corruption of an entire partition or drive." The potential for data errors due to software embedded on certain Pentium computer chips has also been discovered.[1]

1.2. Why Is This Important?

Fig 3. **SOFTWARE'S IMPORTANCE**

-Tremendous growth in use of software.
-Growth in use of software to control critical systems (life supports, safety systems, aircraft, nuclear power plants, etc.
-Mechanical interlocks are being replaced with software interlocks.
-Lack of discipline in generating software now exists.
-Many critical accidents have been associated with software.
-Growth in use to continue.

There has been tremendous growth in use of software to control systems. Software has been used to control critical life-support systems as well as flight controls on military and civilian aircraft. Mechanical interlocks which prevent unsafe conditions from occurring (such as disabling power when an instrument cover is removed) are being replaced with software controlled interlocks. At times a lack of discipline in generating software has existed.

Fig 4. **SOFTWARE'S IMPORTANCE** (Continued)

DEMAND, RISK and NEEDS:
General Bernard Randolph:
...demand for software used to control military and aerospace is growing at 25% per year....
...[software's] cost and schedule growth are due to "a failure of systems engineering and the requirements process."
Critical to weight savings in systems.
Critical to eliminating the need of personnel who could be used better elsewhere!

This growth will continue. General Bernard Randolph said demand for software used to control military and aerospace is growing at 25% per year and that cost and schedule growth are due to "a failure of systems engineering and the requirements process."[2] The size of the software also continues to grow. From a "few" lines of code twenty years ago to 500,000 source-lines-of-code (SLOC) for only the flight software of the Space Shuttle[3] and 1.588 million SLOC for the F-22 Fighter.[4]

This software is critical to weight savings in systems. The use of a computer system to control aircraft and spacecraft has tremendous weight and cost advantages over conventional electro-mechanical systems and has led to its rapid use and acceptance. Software use is also critical for eliminating personnel who could be used better elsewhere.

The application of software in the automotive industry has gone from an eight bit processor controlling engine applications to a power PC to add more and more built-in diagnostics, suspension controls, etc.

However, some problems have become apparent. There are many potential and unrecognized pitfalls to the application of

software that are only now being realized. Because of the complexity of software, it has been cited that only 1% of major software projects are finished on time and budget and 25% are never finished at all.[5] Also, people treat software controls as a black box and often have not attempted to predict the reliability and safety implications of their software.

Many serious incidents in safety-critical applications may have been related to software and the complex control interfaces that often accompany software controlled systems. One example occurred when, "in 1983 a United Airlines Boeing 767 went into a four-minute powerless glide after the pilot was compelled to shut down both engines." This was due to a computerized engine-control system (in an attempt to optimize fuel efficiency) ordering the engines to run at a speed where ice buildup and overheating occurred.[6]

A China Airlines A300-600R Airbus crashed in part because of cockpit confusion. "Essentially, the crew had to choose between allowing the aircraft to be governed by its automatic pilot or flying it manually. Instead, they flew a half-way measure, most probably because they failed to realize that their trimmable horizontal stabilizer (THS) had moved to a maximum nose-up deflection as an automatic response to a go-around command. It was defeating their effort to bring the aircraft's nose down with elevator control ...."[7]

Because of these problems we need to ask the following questions: What computer system errors can occur? What are the risks to the system from software? Why do accidents involving software happen--from both the systems engineering and the software engineering viewpoint? What are some software reliability or (safety) axioms that can be applied to software development? How can we be aware of the real risks and dangers from the application of software to a control and sensor problem? How can the design of software be improved?

1.3. Software Quality

Part I and II of this tutorial, "Software Benefits And Limitations" and "Software Quality And The Design And Inspection Process," will answer a number of questions.

Fig 5. **SOFTWARE TOPICS**

What are some useful software quality metrics? What tools exist to improve software quality? What should specifications for software contain? How are the quality and reliability of software assessed? What would you specify to improve software safety? What are the tools that affect software reliability and how do they affect software quality? What are factors that affect tradeoffs and costing when software quality is evaluated? How do you improve software quality? Software quality should also be defined in terms of correctness, interoperability, flexibility, efficiency, validity and generality. This will also be discussed.

1.4. Software Safety

Because of the often catastrophic effects of software errors, software development is now a key factor affecting system safety. Therefore, a system can only be safe if its software can not cause the hardware to create an unsafe condition. Software safety is the effective integration of software design, development, testing, operation and maintenance into the system development process. A safety-critical computer software component (SCCSC) is a computer software component (processes, modules, functions, values or computer program states) whose errors can result in a potential hazard, or loss of predictability or control of a system. System functions are safety-critical when the software operations that, if not performed, performed out-of-sequence or performed incorrectly could result in improper control functions that could directly or indirectly cause or allow a hazardous condition to exist. How can this software be improved?

2. *OVERVIEW: HOW DO FAILURES ARISE?*

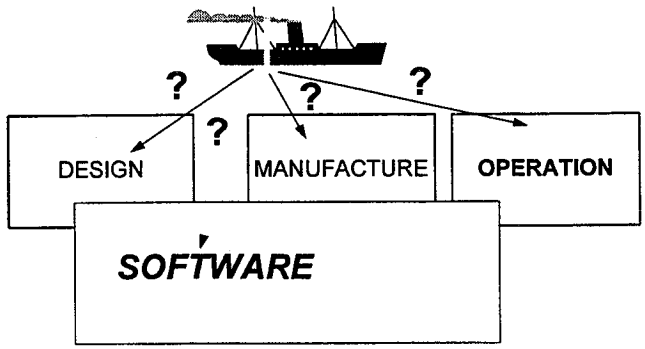


Fig. 6 -- Failure Origins

Generally, we can say that all failures come from the design process, the manufacturing process or operation of the equipment and in this case the computer, its associated software and the system that it controls. Software is becoming a critical source of failures--critical because failures often come in previously unexpected ways. In the design of mechanisms or structures, through a long history of the design process the type and severity of failures have become well known. Hardware failures can often be predicted, inspections can be set up to look for potential failures and the manufacturing process can be changed to make a mechanical system more reliable.

In a mechanical system a small anomaly or error in the design or operation of a system often produces a predictable and corresponding failure. Software is different. With software, an incorrect bit, a corrupted line of code, or an error in logic can result in disastrous consequences. Testing to validate a mechanical system (though not perfect) can be set up to validate "all" known events. On the other hand, software with only a few thousand SLOC may contain hundreds of decision options with millions of potential outcomes that cannot all be tested for or even predicted. Also, historically the design and behavior of mechanical systems have been well known. Expanding the performance envelope of the design led to a new system that was similar to the old system. The behavior of the new mechanical system was predictable. This does not happen with software. With software, minor changes in a program can lead to major changes in output.

2.1. Error Types

What are the types of errors that can occur with a computer controlled system and where do they come from? There are many sources of error (see Figure 7):

A hardware failure can occur in the computer itself, like any other electrical device.

Hardware logic errors (in program logic controllers (PLCs)) can be caused by mistakes in design or manufacture.

Coding errors can occur in the program or the program can become corrupted.

Requirements errors: missing, incomplete, ambiguous, contradictory or incomplete specifications.

Logic errors in the program code can for a given set of inputs, cause the program to reach a state that was never intended.

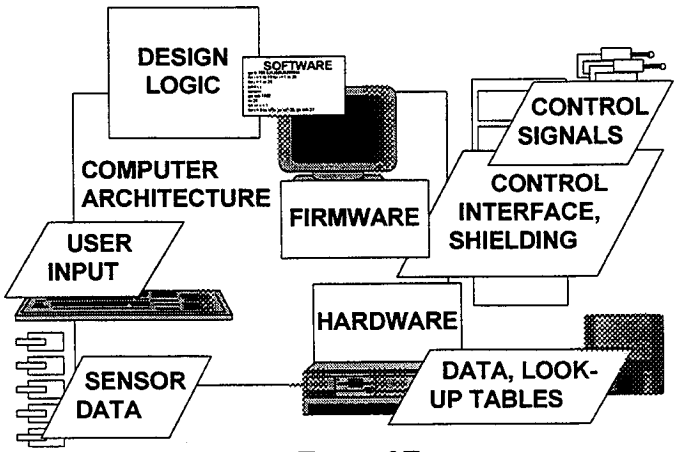


Fig. 7-- Types of Errors

Corrupted data from partially failed sensors or internal look up tables can have errors.

User interfaces problems can be extremely difficult to find (e.g., multiple points to turn off computer control of a system, or keyboard buffers that are too small).

Faulty software tools (e.g., finite element structural analysis code generation programs) with faulty logic and outputs.

The architecture of the computer can vary from platform to platform and cause problems. Software verified on one platform often behaves differently on another platform.

The interfaces between computers or computers and sensors can be faulty or difficult to use.

2.2. Hardware /Software Failure Differences:

Fig 8. **HARDWARE & SOFTWARE FAILURE DIFFERENCES**

-Differences in methods of reliability prediction, inspection, testing between software and hardware components.

-Due to nonphysical abstract nature of software which are not based on cumulative damage.

- As a discipline, software reliability uses few of the tools and methods that apply to hardware reliability.

There are vast differences between the methods used for prediction, inspection, testing and verification of reliability in software versus methods for system hardware components. This is due to the nonphysical, abstract nature of software, the failures of which are almost always information design oversights or programming mistakes and are not caused by environmental stresses or cumulative damage. Furthermore the design rules for mechanical systems are usually well known. A vast amount of historical data on similar systems is available. Mathematical models of wear, fatigue, electrical stress, etc are available to make life predictions. Each software system is often unique. Even with some code reuse, complexity makes reapplication difficult. Some features of software reliability compared to hardware reliability are given in Table 2.2.

3. *TYPES OF SOFTWARE*

3.1. Based On Timing & Control

Fig 9. **TYPES of SOFTWARE Based On Timing/Control**

-What is the allowability of real-time human assessment of the system.

-What is the allowability of real time human interference with the system.

-Is the software autonomous or informational.

-Is the software time-critical or non time-critical.

-Is the information provided general or of a critical nature?

Software risks and impact on systems and data can be evaluated based on what and how the software interacts with a system, how humans interact with the system and the software and whether or not this activity is carried on in real time. Questions to be asked are: (1) Does the software control a system or does it just provide information? (2) Is real-time human evaluation of output and interference allowed? (3) Is the software output time-critical or non time-critical?

Table 2.2 -- Hardware and Software Failure Differences

Category	Hardware	Software
Reliability Prediction Tools	Many mathematical models exist for predicting wear, fatigue life and electronic component life.	Reliability predictions are nearly impossible due to the non-random distribution of errors.
Causes of Failures	Wear-out, misuse, inadequate design, manufacture or maintenance or incorrect use can contribute to failures.	Poor design affects software (The computer system on which the software resides can also fail).
Redundancy	Hardware reliability is usually improved with redundancy.	Software reliability (except possibly for multiple voting systems) is not improved with redundancy.
Hard or Soft Failures	Soft failures (some degradation in service before complete failure) often occur due to wear, chemical action, electrical degradation, etc.	Usually no soft failures occur (However, there may be some recovery routines that can take the system to a safe state, etc.)
Maintenance	Usually testing and maintenance improve hardware and increase reliability.	Software reprogramming may introduce new and unpredictable failure modes into the system. Reliability may be decreased. Any change to the code should require complete retesting of the software, but this is usually not done.
Reliability Prediction Methodology	Design theory, a history of previous systems and load predictions all allow excellent reliability predication.	Software reliability is a function of the development process

(4) Is the data supplied by informational software critical or non-critical? These issues are summarized in Table 3.1. Also reference MIL-STD-882C, System Safety Program Requirements, where types of software are based on levels of control and hazard criticality.

3.2. Based on Run Methodology or Environment.

Fig. 10. **TYPES OF SOFTWARE Based On Environment and Type of System Controlled**

BASED ON ENVIRONMENT:
-Interactive
-Batch.
-Remote Job Entry.
BASED ON TYPE OS SYSTEM CONTROLLED:
-Embedded Software
-Applications Software
- Support Software

Another classification methodology of software is based on how it is run:

Interactive implies a program that is continuously running and interacting with the operator.

Batch implies a single run or process of a program (often acting on data--such as a finite element analysis) where a single output will occur.

Remote job entry implies a software environment where programs are submitted or started by others, (at remote locations) again usually for a single output.

Finally, software may be classified according to the environment in which the software operates. For example:

Embedded software is computer code written to control a product; it usually resides on a processor that is part of the product. Typical applications of embedded software include boiler controllers, washing machine computer controls, automobile computer control, etc.

Applications software includes programs to analyze data. It often runs as a batch job on a computer with limited input from the user once the job is submitted. Typical applications include payroll systems, finite analysis programs, material requirements planning (MRP) systems (updating sections).

Support software tools may be thought of as another class of programs. They are used to develop, test and qualify other software products or to aid in engineering design and development. Examples are compilers, assemblers, Computer Aided Software Engineering Tools (CASE), etc.

4. EXAMPLES OF COMPUTER SYSTEM ERRORS

What are some examples of the problems that have been observed with the application of software to control processes and systems?

Fig. 11. **EXAMPLES of COMPUTER SYSTEM ERRORS**

RADIATION MONITOR
-Timing problem with data entry
-Hardware interlocks removed.
CHEMICAL PLANT
-Programmers did not understand process.
SPACE SHUTTLE
-Software revisions were not rechecked.
AIRLINER
-Personal compute shuts down navigation system

Here are some additional examples:

SPACE PROBE: Clementine 1, which successfully mapped all of the Moon's surface was to have a close encounter with a near-Earth asteroid. A hardware or software malfunction on the spacecraft "resulted in a sequencing mode that triggered an opening of valves for

Table 3.1 -- Classification of Software Based on Level of Hazard and Control

Software Control	Information	Human/Other Control Interference	Real Time	Examples
Autonomous control exercised over hazardous systems.	Some information may be available but insufficient for real-time interference.	May be possible but not desirable. *Often no other independent safety systems.*	Yes	Space shuttle main engine and solid rocket booster ignition sequence.
Semi-autonomous control exercised over hazardous systems.	Real time information is available to allow human/other system interaction and control.	Possible and desirable under some circumstances. *Other independent safety systems or ability to disengage.*	Yes	Aircraft terrain following system, medication dispensing device, nuclear power plant safety systems, automatic go-around mode in aircraft (override).
Mix of computer and human control over hazardous systems.	Real time information is available to allow human interaction and control. Human control of some functions.	Yes, required for some sub-systems of operation. *Other independent safety systems.*	Yes	Aircraft fly-by-wire system of unstable aircraft (example B-2) where computer translates pilots control requests into feasible flight surface modifications.
No, but generates information requiring *immediate* human action.	Complete real time information presented to allow human control over hazardous systems.	Human interaction required to properly control the system. *Other independent safety systems.*	"Yes"	Aircraft collision avoidance systems, nuclear power plant instrumentation, hospital patient vital signs.
No, but human action based on information.	Information presented non-real time. Software *does* provide *critical* information.	Human actions and decisions are directly influenced by information. *Other checks.*	No	Statistical process control information of machine tools, historical medical information summaries.
No, but human action based on information.	Information presented non-real time. Software *does not* provide critical information.	Human actions and decisions are directly influenced by the information.	No	Financial and economic data.

four of the spacecraft's 12 attitude control thrusters, allowing all of the hydrazine propellant to be used up."[8]

CHEMICAL PLANT: Programmers did not fully understand the way a chemical plant operated. The specifications stated that if an alarm occurred, all process control settings were to be frozen. The resulting computer system released a catalyst into a reactor and began to increase cooling water flow to the reactor. While the flow was increasing the system received an oil sump, oil low alarm and froze the flow of cooling water at too slow a rate. The result was that "the reactor overheated and the pressure release valve vented a quantity of noxious fumes into the atmosphere."[9]

SPACE SHUTTLE: An aborted mission nearly occurred during the first flight of Endeavor to rendezvous and repair an Intelsat satellite. The software routine used to calculate rendezvous firings "failed to converge to a solution due to a mismatch between the precision of the state-vector tables, which describes the position and velocity of the Shuttle."[10]

AIRLINER: A laptop computer used by a passenger on a Boeing 747-400 flying over the Pacific caused the airliner's navigation system to behave erratically. When the computer was brought to the flight deck and turned on "the navigation displays went crazy."[11]

5. *SOURCES OF ERRORS*

Where do software errors come from. Rather than just concentrating on concern for errors in the software logic, the investigation as to sources of problems needs to be expanded. Anytime an analog and/or electro-mechanical control system is replaced with a computer system besides many unique problems can occur.

5.1. Organizational Problems.

How do errors occur, what causes them and how can they be eliminated? What are some of the procedures, organizational arrangements and methodology that cause problems with software?

- **Communication** — **Silver Bullets**
- **Documentation** — **Personnel**
- **Standardization** — **Software Reuse**
- **Configuration Management**

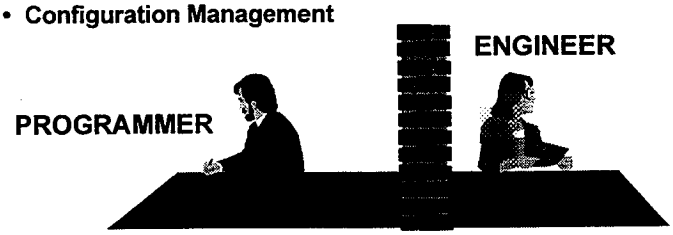


Fig. 12 -- Sources of Errors: Organizational Problems

(1) Communication:

Often there is a lack of communications and understanding between the software programmer and the system or design engineer. The designer does not understand the software and the programmer does not truly understand the system with all its potential failure modes (they do not have domain specific knowledge). Programmers frequently fail to understand the potential for problems if certain things are not done in a logical sequence. For example, "start heater and add fluids to boiler" may be "logical" programming sequences, but what if the computer has a fault after the heater is started, before enough fluid is added to the boiler?

Design or safety engineers frequently do not understand software and how it will control the system and the potential

for software problems. Often the computer and its software are treated as a black box with no regard for the consequences if the unit fails. In the past system safety engineers have ignored software or looked at it superficially in analyzing systems.

(2) Documentation:

There is a lack of software documentation standards, testing and verification procedures. Practices such as not documenting the software analysis, inspection and test process or last minute fixes without retesting and reverification cause many problems. Lack of design and verification tools may exist. Formal procedures for software inspection may be lacking or the procedures may be in place but essentially ignored by the software development group.

A potential flight problem was noticed on one experiment scheduled to fly in space to evaluate the effects of microgravity. To correct it the software was changed during a preflight checkout on a holiday. The change was not verified. During the mission, the heaters on a device would develop only 25% of the needed power. The simple software change caused a the loss of some science data.

(3) Standardization:

There is a lack of software structure standardization in many organizations. Not requiring adherence to software standards is an underlying contributor to many system failures. Trying to be elegant in writing software, using complex techniques and neglecting internal comments and written documentation can seriously affect the quality of software. Lack of structure standardization also lessens reuse of software as well.

(4) Personnel:

There is little attempt to keep good programming talent. A turnover results in a loss of corporate knowledge. Changes in personnel reduces reuse of code and causes problems when maintaining software as well.

(5) Silver Bullets:

Over reliance on silver bullets to solve all a company's software problems causes real issues to be overlooked. One of the most difficult problems to deal with is unrealistic hope that some advance in software development technology, some new code generating tool or object oriented super code will make software generation problems go away. This also manifests itself when these state of the art techniques are exclusively relied upon instead of using good documentation, formal requirements and continual interface between software, design and safety personnel.

(6) Configuration Management:

There is often a lack of control over software changes both during development and during maintenance of software. Unauthorized changes in software or undocumented changes in software put in by a programmer to fix a possible mistake may cause many problems downstream. Toward the end of a project pressure to just get the job done encourages code changes without proper review or documentation.

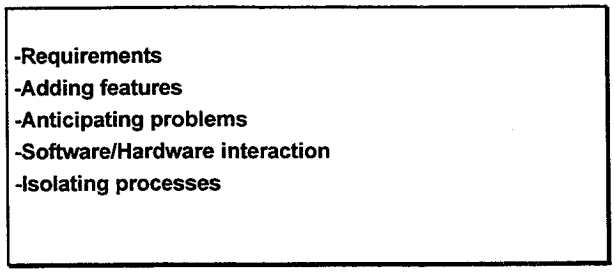
(7) Software Reuse:

There is little attempt to reuse software. Many software programs are started from scratch (again with little control over how the code is to be written). Note that attempts to reuse code are often disastrous because of unknown defects.

5.2. Design and Requirements Problems

Besides organizational problems, poor analysis and flowdown of requirements' specifications for an individual project can cause errors, delays and cost overruns.

Fig. 13. **SOURCES of ERROR: Design & Requirements**



These problems include:

(1) Requirements:

Poorly defined requirements for a specific software project can cause a cost overrun and increase the probability that code logic errors will be introduced. When real-time systems are developed for new applications or applications outside the normal areas of expertise of the software engineers, the need for additional requirements to implement the basic system are often needed. They are frequently discovered while the software development process is well underway. Requirements are often inconsistent, incomplete, incomprehensible, contradictory and ambiguous as well.

(2) Additional Features:

Adding new features to the software is also major problem. There is a perception that requirements for new features can continue to be added long after programming has started with little negative effect. But adding to performance requirements has bad effects on system software. This addition of many new requirements as the project progresses may be viewed as a trivial problem by the design engineer but the software must be changed for each new requirement. This adds to the risk of increasing errors in the function or logic. The question needs to be asked: Have the requirements been analyzed as a complete set?

(3) Anticipating Problems:

There is often little attention given to protecting the software controlled system from off-nominal environments. All the emphasis is put into fulfilling performance requirements without a careful analysis of what can go wrong with the

system. Little attention is paid to what states the system can reach through an unanticipated series of events.

(4) Software/Hardware Interaction:

There is often a lack of understanding of how the program will actually run once a system is operational. There may be problems with processing all the sensor data during a clock cycle. The software may be unable to deal with changes in physical conditions and processors.

(5) Isolating Processes:

Putting too many unnecessary software processes on a computer controlling a safety-critical system can reduce assurance that critical processes will be handled properly (safety-critical refers to systems whose failures can cause loss of life, loss of mission or loss of system).

5.3. Other Problem Areas

Problems with the software code are not our only concern. The associated hardware, sensors and interfaces can also pose special risks. Incorrect data, the reliability of the system itself and the production, distribution and maintenance of software are also problems.

Fig. 14. **SOURCES OF ERROR: Other Areas**

- **Reliability**
- **System/Sensor Interfaces**
- **Radio Frequency (RF) Noise**
- **Maintenance and manufacturing problems.**

(1) Reliability:

The reliability and survivability of the computer hardware, sensors, and power supplies are often not adequately planned for. The Central Processing Unit (CPU), memory or disk drives of a computer can fail. The system can lose power. Excess heat or voltage spikes can also cause unanticipated errors in performance, output or complete system shutdown.

(2) System/Sensors Interfaces:

The interfaces between sensors and other mechanical devices can fail. Cables can become damaged and power supplies to sensors or servo-controllers can fail. Often the anticipation of these events and effective solutions are not handled adequately.

(3) RF Noise:

The effect of radio frequency (RF) noise on computers, on signals from sensors, and on components with damaged or incorrect grounding and shielding is often not anticipated. RF noise can affect the operation of a computer processor, its memory and input/output devices as well. RF noise can also affect sensors, poorly shielded cables, connectors and interface boards (e.g., fiber optic to digital conversion, etc.). This can cause errors or erroneous readings.

(4) Manufacture and Maintenance:

Proper manufacture, reproduction and distribution of software are not always handled properly. This results in compilation errors and improper revisions of code being distributed. Integration problems can occur during the assembly of code, linking program modules together and transferring files. Poor control over maintenance upgrades of software and firmware also causes problems. Errors can result from improper loading of programs, wrong batch files and patching to the wrong revision of software, etc.

Another classification of errors comes from a Rome Laboratories study and is shown along with their percentages of occurrence in Table 5.3. The study also shows the importance of interface design and documentation.[12]

Fig. 15 -- Sources of Errors by Percentage

Source of Error	Percent
Logic	21.29
Input/Output	14.74
Data Handling	14.49
Computational	8.34
Preset database	7.83
Documentation	6.25
User Interface	7.70
Routine to Routine Interface	5.62

6. *TOOLS TO IMPROVE SOFTWARE SYSTEM RELIABILITY & SAFETY*

For each of the aforementioned problem causing agents, there is a way to minimize risk and even eliminate to problem.

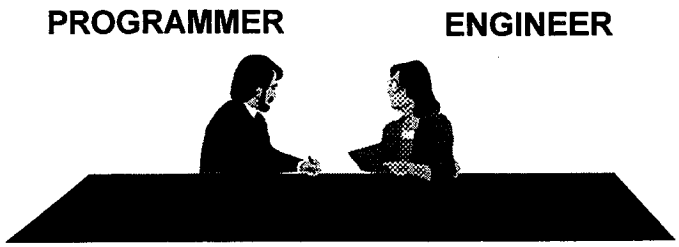


Fig. 16 -- Tools to Improve Software System Reliability & Safety

They are as follows:

6.1 Organizational Improvement

Various tools and techniques (some of which have been briefly mentioned) when properly applied and supported at all levels of the organization can do much to improve reliability and safety of software.

(1) Communication:

Improve communication between designers, software engineers and safety engineers through concurrent engineering and safety review teams and joint training. Concurrent engineering with regular meetings between design and software engineering to review specifications and requirements will do much to improve communications. Continuous discussions with the end users will also improve understanding of the background of the various system performance requirements. Joint training and cross training will encourage developing informal relationships and informal communication. Software safety review committees made up of design, software and safety personnel who continually meet to review specifications and implementation of software will help to assure that safety-critical software performs properly. Also, specifications have to be carefully written, not just in "legal" terms but clearly describing how the system should work. This will convey the maximum amount of information.

(2) Documentation:

Improve software documentation standards, testing and verification procedures. Encourage the application of standands for all software projects. This includes general requirements for all system development projects, which industry or military standards will be followed and what specific documents are to be generated for any specific product. These documents may include a software management plan, a software assurance plan, a software configuration management plan, software requirements' specifications, a software test plan and a software version description document (see Section II for more details).

(3) Standardization:

Set and enforce software structure standards. The software must be structured with good specifications as to what is and what is not allowed. The programmer should not design some "clever" program that cannot be readily understood or debugged. Enforce safe subsets of programming language, coding standards and style guides.

(4) Personnel:

Provide incentives to keep good programming talent and maintain the corporate knowledge base. There should be a mix of programming skills & experience. The ability to transmit *practical* programming knowledge to new programmers who only have classroom training with little or no insight into real world problems is very important. Keeping senior programmers or senior managers who can review software and participate in Independent Verification and Validation (IV&V) of software across missions or products is also of real benefit. Efforts should also be made to retain workers who know the software systems to support software maintenance and new applications of the code. Also, provide training in proper methodologies.

(5) Silver Bullets:

If the introduction of major changes in the procedures for generating software is contemplated, they must be review with great care. Their impact on the software personnel, maintenance of software and software standardization must be evaluated carefully. Projects already underway and projects scheduled to be started may or may not benefit from the change. Major disruptions to personnel can result. As with any major change in the way a product is designed and developed, careful and complete training of personnel, a free flow of information on the new system, assurances as to support of existing programmers and gradual introduction of the new methods (e.g., starting on one small project, etc.) is required.

(6) Configuration Management:

Implement consistent controls over software changes and the change approval process. This can be accomplished with a variety of software development products including software configuration management and code generation tools. Computer Aided Software Engineering (CASE) tools and other configuration management techniques can automatically compare software revisions with previous copies and can limit unapproved changes to software. Other programming tools provide mission simulation and module interface documentation.

Software should also be modularized to facilitate changes and maintenance. The software modules should have low coupling (the number of links between modules shall be minimized) and the software modules should have high-cohesion (the level of self-containment).

Use a "clean room approach" to develop software. This implies a highly structured programming environment and tight control of the specifications for the software and system. It also implies support and adherence to the software analysis specifications.

(7) Software Reuse:

Encourage reuse of software with strict controls over software structure and procedures for code reuse. Software modules/software reuse also improves reliability. Reused code benefits from faults removed in prior usage. Modularized software with well documented and verifiable inputs and outputs also enhances maintainability. Lewis Research Center's (LeRC's) launch vehicle programs are reused for each mission with only minor modifications. This has achieved excellent reliability results.

6.2. Design and Requirements Improvements

The hardware and the software must be integrated to work together. This integration includes the entire system with input sensors and signal conditioners, analog to digital (A/D) boards, the computer hardware/software itself and the output devices (control actuators, etc.). Basic design methodology

can improve software as well. Some basic approaches supporting this concept are as follows:

(1) Requirements:

Spend sufficient time defining and understanding requirements. The system, software and safety engineers should spend an adequate amount of time working with the end user to both develop requirements, be able to express the requirements in mutually understandable language and to have requirements that are testable and verifiable.

(2) Additional Features:

Limit changes in requirements once the software design process starts. The question needs to be asked: Is this additional really a necessary requirement? Instead, functionality should be reduced if necessary to achieve safety and basic performance goals. A huge number of ancillary, non-critical devices and special graphical user interfaces may not be necessary and only complicate and slow the system.

Put software in its proper place of importance. Many people over confidently think, if a computer with its software is controlling a system, it can never fail. They will believe computer controlled readouts instead of their own senses. This has given people a false sense of security.

(3) Anticipating Problems:

Fully analyze all the ways the software controlled system can fail. What undesirable states can the system reach? Once this is done procedures and methods can be implemented to make sure these undesirable states and failure modes cannot be reached. Make sure that they are not attainable through some unusual (though not impossible) combinations of software states, environment and /or input data. Then the system will not be vulnerable to these failures.

While software does not degrade, it is virtually impossible to prove the correctness of large, complex, real-time systems (however, selective use of logic engines can be effective in reducing uncertainty about a systems performance). Error detection, correction and recovery software development are also necessary to achieve fault tolerance. Examples of common errors include inconsistent data in databases, process deadlock, starvation and premature termination, run-time failures due to out-of-range values, attempts to divide by zero and lack of storage for dynamically allocated objects.

Software should detect and properly handle run time errors. Software controls should assume the worst and prepare for it. What undesirable states can the computer get to and how can each of these states be prevented. A careful analysis of responses to failed or suspect sensors should also be made.

Software capable of real-time diagnosis of its own hardware and sensors is very useful. Memory can be protected with parity, error correcting code and read-only circuitry in memory. Messages received should be checked for accuracy, and routes can be automatically changed when errors are detected. Predefined system exceptions and user defined fault exceptions should be designed into application software. Predefined exceptions can be raised by run-time systems. The software should also have built-in or operating system recovery procedures. Information for recovery includes processor id, process name, data reference, memory location, error type and time of detection.

(4) Software/Hardware Interfaces:

Computer timing problems and buffer overload problems must be eliminated. If all alarms and sensors cannot be read in one clock cycle of the Central Processing Unit (CPU) errors may occur or alarms may be missed. Overloaded buffers can result in CPU "lock-up."

Load balancing should be a part of operating system software routines because failures are often caused by overloading one or more processors in the system. This can be caused by an increase in message traffic or the inability of a processor to perform within time constraints, etc. Dynamic traffic time sharing where message streams are distributed among identical processors with a traffic coordinator keeping track of the relative load among processors is another potential tool to support complex systems.

(5) Isolating Processes:

Systems for safety critical applications need to be separate from everything else. System specifications often require gathering data from hundreds of sensors, and performing all sorts of non-critical tasks. Segregating these non-critical tasks to a separate computer system will often improve assurances that safety critical functions will be not be disrupted by defects in non-critical resources. Safety-critical modules should be "firewalled."

For critical systems use proven hardware and technology. Older computer systems and software that are "flight-proven" and do the job should be chosen over newer computers whose standards are rapidly evolving where critical applications are involved.

Analog interlocks on safety critical systems should be replaced with software interlocks only with the greatest of care. A thorough, well-documented analysis of what would happen with a computer failure and the system failure that the interlock protects should also be made. An example of the problem of replacing mechanical interlocks with software interlocks involves a radiation therapy machine. An early model of the therapy machine had a hardware interlock to prevent radiation overdoses. The interlock was removed on a later model and replaced with software logic. Several people were killed when the machine overdosed them with radiation. The problem was caused by the operator interface, poorly documented data input procedures and inadequate safety procedures. The earlier model never experienced the problem since the program did not control the interlock.[13]

In many cases, safety critical systems can have an analog process (or a stand-alone computer) capable of taking over if the primary computer fails. If a computer control fails on a

process plant, an analog backup system (which was presumably being controlled by the computer) could keep the process running (though at less than optimum conditions). Alternatively, control actuators could go to a safe position if a failure occurred. Usually, the process must be allowed to proceed to some nominal conditions (e.g., partial cooling water, partial product inflow into a process, etc.) before shutting down.

Monitor the health of the backup systems and the output of software control commands independently of the main control computer. Have a separate computer performing health checks on the main computer and on safety critical sensor outputs.

Special tests should be done to verify the performance of safety critical software. This includes testing to verify that the software responds correctly and safely to single and multiple failures or alarms. The software should properly handle operator input or sensor errors (e.g., data from a failed sensor). Tests should be made to assure that the software does not perform any unintended routines. Detection and action upon failures with respect to entry into and execution of safety-critical software components and the ability to receive alarms or other inhibit commands should also be provided.

Formal methods can use abstract models and specification languages to develop correct requirements. Logic engines can be used to prove correctness of the requirements.

Lewis Research Center's (LeRC's) launch vehicle program has for many years verified the software for each mission by running the complete program in the mission simulation lab. All of the mission constants and components are checked and verified. LeRC has never lost a vehicle due to software problems.

6.3 Other Improvements

The hardware/software system must also be integrated with input sensors and signal conditioners (analog to digital boards, etc.) and the output devices (e.g., servo-controlled actuators). Reliability of all this hardware is also an issue. Some basic approaches to total system performance are as follows:

(1)Reliability:

The reliability and survivability of the electronic components associated with the software control system can be improved with proper protection of components from vibration, excess heat and voltage and current spikes. Properly *maintained* grounding and shielding also must be assured with maintenance training and documentation. Having robust sensors, actuators and interfaces will also contribute to a more reliable system. Sensor failure can cause wrong data to be processed. Even the fraying of cables has been linked with possible uncontrolled changes in aircraft flight surface actuation. The reliability of computer controlled output devices (servo-actuators, valves, relays, etc.) must be verified as well. The output devices may also subject to noise problems. Error recovery and restart procedures should be included in software and properly tested.

Passive controls should be designed so that failures cause the system to go to a safe state. If input commands or sensor readings are suspect, the system should go to a safe condition. The latter should be done (as previously noted) with an analog backup or an autonomous software module. The autonomous software module should be in a separate backup system.

Multiple Voting Systems (multiple computers running the same task in parallel with independently written programs) might help improve reliability.[14] Multiple computers with software written for the same functional output but developed independently is one way to handle the critical problem of software getting to some condition that was never intended. Systems should sense the occurrence of anomalies and alert the operator. Health monitoring of the controlled system as well as the computer itself, frequent self checks, etc, should be included in the program.

Redundant systems need to have separate power sources and locations (to avoid common mode failures). Use uninterrupted power supplies for critical software systems. Have battery backup for as long as is needed to switch to manual operation. Avoid a common power supply that can send a surge to all devices at once or can shut off all devices at once.

A distributed system can also be used to improve reliability. The system can sense problems in one processor and will transfer its work to another processor or system. Hardware components degrade with time and represent the most important factor in ensuring reliability of real-time systems.

Note however that the complexities of a distributed system can cause new problems and possible reduce reliability. For example synchronization and precision of numerical values between programs and communications procedures can cause errors. More resources are also consumed for coding and testing and programs become larger (with more chance for error).

(2) System/Sensors Interfaces:

The computer and sensor interfaces must be thoroughly tested to prevent mechanical failures, intermittent contacts, connector problems and noise. Again, provisions for data out of acceptable ranges must be made.

(3) RF Noise:

Radio Frequency (RF) noise problems can be avoided. Input and output data should be validated before use. The software should check for data outside of valid ranges and take appropriate action such as an alarm or system shutdown, etc. Proper maintenance procedures and training in the removing and replacing of grounding and shielding should be developed. The interaction of and possible need for separate analog and digital grounds should also be investigated. Thorough testing of the system in all anticipated environments should of course be performed.

(4) Manufacturing and Maintenance:

The duplication, loading and maintenance of software must be planned for and controlled. Proper procedures must be developed to assure that the proper code is loaded on each model of processor. Proper verification of all new compilations of code must also be performed. Buggy compilers can introduce defects. Subtle changes from one revision of an operating system to another can cause subtle changes in response to the same code. Procedures and requirements for maintenance upgrades must also be made. The updated software should be adequately tested and verified (to the same level, extent and to the same requirements as the operating software) for accuracy (performance), reliability, and maintainability. New software should be modularized and uploaded as individual modules when maintenance is being performed. Also, whenever possible, issue firmware changes as fully populated and tested circuit cards (not as individual chips).

7. *SOFTWARE DEVELOPMENT TOOLS*

There are a number of methods that can be used to analyze and verify software.

Fig. 17. **SOFTWARE ANALYSIS TOOLS**

- **Fault tree analysis**
- **Petri net analysis**
- **Hazards analysis**
- **Formal logic analysis**
- **Software failure modes and effect analysis**

Some of these include:

Fault tree analysis can identify critical faults. Potential faults or problems are identified and then all the conditions that can lead to those faults are considered and diagrammed.

Petri net analysis provides a way of modeling systems graphically. A Petri Net has a set of symbols that show inputs, outputs and states with nodes that are either "places" (represented by circles) or "transitions" (represented by vertical lines). When all the places with connections to a transition are marked, the net is "fired" by removing marks from each input place and adding a mark to each place pointed to by the transition (the output places).[15]

Hazard analysis can also be performed on the system. Formal methods for identifying hazards can be used to evaluate software systems.[16]

Formal logic analyzers are logic engines that can verify specifications. Some source analyzers can reveal logic problems in code and branching problems.

Pseudo codes are similar to programming languages but are not compiled. They are used for program design and verification. They have flow and naming notation of the programming language but with a readable style to better understand program logic.[17]

State transition diagrams (STDs) are also a useful tool. STDs are graphs that show the possible states of the system as nodes and the possible changes that may take as lines. This can highlight poor architecture or unnecessarily complex computer code.[18]

Software Failure Mode Effects Analysis (FMEA) analyzes what can go wrong with the software and what can go wrong with the system itself. The FMEA should analyze whether or not the system is fault-tolerant with respect to hardware failures and to make sure the system specifications are complete. The actual failure of the computer hardware usually results in a hard failure and the effects are easily identified. The effects of failures handled by software may not be so clear. For example, how does the software handle the loss of one piece of sensor data or a recovery from a fault?

8. *SOFTWARE SAFETY AXIOMS and SUGGESTIONS*

These axioms are food for thought. They should be looked at, read and reread. The principles behind them should be thoroughly understood.

Fig. 18. **SOFTWARE SAFETY AXIOMS**

- **Persons who design software should not write the code and those who write the code should not do the testing.**
- **Accidents are caused by incomplete or wrong assumptions about the system or process being controlled. Actual coding errors are less frequent perpetrators of accidents.**
- **Unhandled controlled system states and environmental conditions are a big cause of "software malfunctions."**
- **Lack of up-to-date professional standards in software engineering and/or lack of use of these standards is a root cause of many problems.**
- **Limit the changes to the original system specifications.**
- **It is impossible to build a complex software system to behave exactly as it should under all conditions.**

Fig. 19. **SOFTWARE SAFETY AXIOMS (continued)**

- **Software safety, quality & reliability are designed in, not tested in.**
- **Upstream approaches to software safety are most effective.**
- **Software alone is neither safe nor unsafe.**
- **Many software bugs are timing problems that are difficult to test for.**
- **Software often fails because the software goes somewhere that the programmer does not think it can get to.**
- **Software systems do not work well until they have been used.**
- **Mathematical functions implemented by software are not continuous functions but have an arbitrary number of discontinuities.**

Fig. 20. **SOFTWARE SAFETY AXIOMS (continued)**

Engineers believe one can design "black box tests" on software systems without the knowledge of what is inside the "box."
Keep safety critical systems as small and as simple as possible; moving any functions that are not safety critical to other modules.
Treat a software control system as a single point failure (often in the past the software was just ignored).
Decide what you do not want to happen -- make sure your program can't get there.
Make sure your system is fault tolerant and that it can recover from faults and instruction jumps.
Use independent verification and validation of software (IV&V).

9. *CONCLUSION*

Software is now used in many safety-critical applications and each system has the potential to be a single point failure or zero fault tolerant (i.e., a single failure will cause failure of the system or if the computer is controlling a hazardous function, a single failure can cause a hazardous condition to exist.

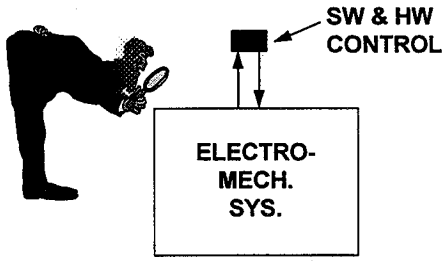


Fig. 21 -- Software Controls on Systems Are Often Ignored

The potential problems with software are not well understood. Computers controlling a system (this includes the computer hardware, the software, the sensors and output devices that direct the flow of energy) are not a black box that can be ignored in safety, reliability and risk evaluation.

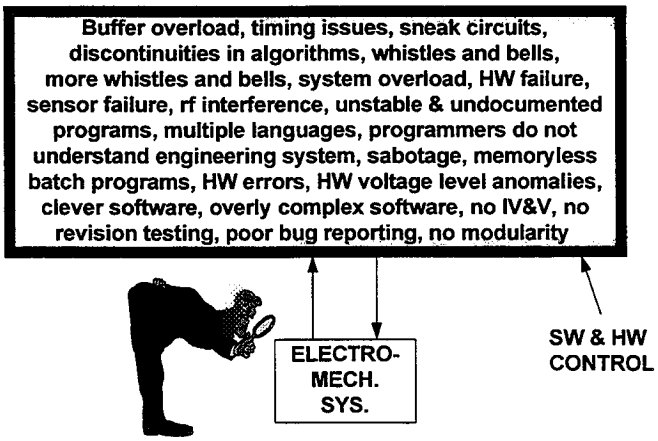


Fig. 22 -- Software Controls on Systems Are a Single-Point Failure

However, if handled properly and applied properly, the use of software and hardware to control a system can be a valuable design option.

There are many ways to improve the software development process. Good communication, documentation, standardization and configuration management benefit the software development process. Correct and understandable requirements are also a major factor in proper software development. Anticipating problems, proper error handling and improving hardware reliability, help to improve confidence in the system as well.

There are many ways to validate and improve software quality (and safety) and this will be discussed in part II of this tutorial -- Software Quality And The Design And Inspection Process.

---

[1] T. Halfhill, The Truth Behind the Pentium Bug, *BYTE Magazine*, March, 1995.

[2] *Aviation Week and Space Technology*, June 19, 1989.

[3] Space Shuttle Flight Software Development Processes, NASA.

[4] "F-22 Software on Track With Standard Process," *Aviation Week and Space Technology*, July 24, 1995.

[5] M. Norris and P. Rigby, Software Engineering Explained, Wiley, 1992.

[6] J. Beatson, "Is America Ready to 'Fly by Wire'?," *Washington Post*, April, 1989.

[7] Michael Mecham, "Auto pilot Go-Around Key to CAL Crash," *Aviation Week and Space Technology*, May 9, 1994.

[8] James R. Asker and Jeffrey M Lenorovitz, "Computer Woes Spoil Clementine Flyby Plan," *Aviation Week & Space Technology*, May 16, 1994.

[9] M. Norris and P. Rigby, Software Engineering Explained, Wiley, 1992.

[10] Space Shuttle Flight Software Development Processes, NASA.

[11] Sharon Begley, "Mystery Stories at 10,000 Feet," *Newsweek*, July 26, 1993.

[12] Chenoweth and Schulmeyer, *Proceedings of IEEE Conference on Computer Assurance*, IEEE Computer Software Assurance Conference, COMPSAC 86, Washington, D.C., 1986.

[13] N. Leveson, Safeware: system safety and computers, Addison-Wesley, 1995.

[14] While this concept is beneficial in theory, some studies suggest that common software logic faults arise from common requirements. See the article by John C. Knight and Nancy G. Levenson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transaction on Software Engineering*, Vol. SE-12, No. 1, January, 1986. Furthermore maintenance and configuration management of this type of system is greatly complicated by having different active versions of code.

[15] M. Norris and P. Rigby, Software Engineering Explained, Wiley, 1992.

[16] N. Leveson, Safeware: system safety and computers, Addison-Wesley, 1995.

[17] M. Norris and P. Rigby, Software Engineering Explained, Wiley, 1992.

[18] M. Norris and P. Rigby, Software Engineering Explained, Wiley, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1997	Technical Memorandum

4. TITLE AND SUBTITLE
Software Design Improvements
Part 1: Software Benefits and Limitations

5. FUNDING NUMBERS
WU-323-93-03

6. AUTHOR(S)
Vincent R. Lalli, Michael H. Packard, Tom Ziemianski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER
E-10609

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-107402, Part 1
IEEE-155NO897-5000

11. SUPPLEMENTARY NOTES
Prepared for The International Symposium on Product Quality & Integrity cosponsored by AIAA, ASQC/RD, ASQC/ED, IEEE/RS, IES, IIE, SAE, SOLE, SRE, and SSS, Philadelphia, Pennsylvania, January 13-19, 1997. Vincent R. Lalli, NASA Lewis Research Center; Michael H. Packard, Raytheon Engineers and Constructors, 2001 Aerospace Parkway, Brook Park, Ohio 44142; Tom Ziemianski, Texas Instruments Inc., Dallas, Texas (work funded by NASA Contract NAS3-26764). Responsible person, Vincent R. Lalli, organization code 0510, (216) 433-2354.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Categories 33, 37, and 38

This publication is available from the NASA Center for AeroSpace Information, (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Computer hardware and associated software have been used for many years to process accounting information, to analyze test data and to perform engineering analysis. Now computers and software also control everything from automobiles to washing machines and the number and type of applications are growing at an exponential rate. The size of individual programs has shown similar growth. Furthermore, software and hardware are used to monitor and/or control potentially dangerous products and safety-critical systems. These uses include everything from airplanes and braking systems to medical devices and nuclear plants. The question is: how can this hardware and software be made more reliable? Also, how can software quality be improved? What methodology needs to be provided on large and small software products to improve the design and how can software be verified?

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software; Reliabilty; Problems; Safety; Improvement; Inspection; Axions	14
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	