Conf-950258--1

SAND94-3112C

A Review of Research and Methods for Producing High–Consequence Software

E. Collins, L. Dalton, D. Peercy, G. Pollock, C. Sicking‡
Software Reliability Working Group
Sandia National Laboratories

ABSTRACT

The development of software for use in high-consequence systems mandates rigorous (formal) processes, methods, and techniques to improve the safety characteristics of those systems. This paper provides a brief overview of current research and practices in high-consequence software, including applied design methods. Some of the practices that are discussed include: fault tree analysis, failure mode effects analysis, petri nets, both hardware and software interlocks, n-version programming, Independent Vulnerability Analyses, and watchdogs.

Techniques that offer improvement in the dependability of software in high-consequence systems applications are identified and discussed. Limitations of these techniques are also explored. Research in formal methods, the cleanroom process, and reliability models are reviewed. In addition, current work by several leading researchers as well as approaches being used by leading practitioners are examined.

INTRODUCTION

The development of software for high-consequence applications creates an urgent need for increased surety (i.e. reliability, safety, security) techniques. This need is a direct outcome of the technological advancements in the computer field and actually began with the introduction of microelectronics. The advent of microelectronics enabled the realization of (micro)processor technology which radically improved the functionality of electronic systems. Coincident with improved functionality was the increased inherent complexity in both (micro)processor hardware devices and their associated software.

While the quantitative reliability of hardware devices has been accounted for in accepted integrated circuit reliability modeling, the associated software does not readily yield to quantitative reliability methods–in fact, other than reliability growth models which have not been proven to be accurate for all applications, no practical quantitative methods exist. Because of this fact, dependence is placed upon testing to arrive at a qualitative level of

‡ We gratefully acknowledge the contributions of Charles Weinstock and Lui Sha of the Software Engineering Institute, Carnegie Mellon University.

This work was supported by the United States Department of Energy under contract DE-AC04-94AL85000.

comfort with respect to the reliability of the software. Quite often, the top-level system reliability represents only the hardware reliability in that the software is assumed not to fail.

Surety in the context of this document represents the full spectrum of concerns related to the behavior of hardware and software-based systems that control "high-consequence" operations. "High-consequence" in this context is intended to be very broad in scope so as to include national security, medical, financial/economic, and information/data applications as they affect human life, and international stability in both peace and competitiveness.

The primary intent of this document is to present to software managers and practitioners a current view of the most promising research, methods, and practices that increase software reliability. It is hoped that the information presented herein will raise the level of awareness with respect to the growing dependencies on very complex hardware and software systems and the resulting risks of accepting such systems. It is further hoped that this increase in the level of awareness will lead to fundamental improvements/changes in approaches to the development of high-consequence systems. Achieving these improvements will involve further development and institutionalization of, software "engineering" methods and practices.

In this document, a brief discussion of various issues motivating the work in this area is presented before the main overview of research and applied methods for producing high-consequence software. The following sections address the current state of the practice and the current state of promising research. Further, reviews of core applications of cited techniques from real-world situations conclude the discussions in each survey section. Conclusions drawn from this sampling of current research and practices for software surety in high-consequence system applications are stated in the final summary section.

PROBLEM STATEMENT

The flexibility of software-controlled computer systems offers significant advantages to designers of systems. Software-controlled computer systems can be configured to distinguish a large number of situations and provide output appropriate to each of them without additional hardware. An analog hardware system cannot duplicate this flexible behavior without additional hardware and the associated costs in money, physical volume, weight, and so forth. Because of the need for this inherent flexibility, software-controlled systems are being deployed in situations where they can cause loss of life, damage the environment, or destroy or damage valuable resources or equipment.

Some examples of these safety-critical applications include control of medical equipment, digital flight control systems for aircraft, automated controls for chemical processing plants, automated transportation systems, and robotics. Software cannot directly cause loss of life but it may control equipment that can cause loss of life if the software behaves in an unanticipated fashion that causes or allows the system to enter an unsafe state.

Software can have unanticipated behavior due to faults present in the software or due to interactions between a hardware fault and the software. Faults can be inadvertently injected into software at any point in the development process; faults can be the result of poorly defined requirements or a failure of the designers to anticipate all possible inputs or use environments. Further, logic faults—an incorrect algorithm, inadequate precision, computational faults, or typographical faults–can be introduced inadvertently during coding. And the tools used to develop the software, such as editors, compilers, linkers, and loaders, may be additional sources that can introduce faults independent of the source code.

To anticipate all possible failure modes (e.g., hardware, software, interface, environment) of a computer system is very difficult, if not impossible. A computer system is a very flexible and complex multi-state machine yielding more possible variations than can reasonably be examined. On top of this complexity, the application software executing on the computer attempts to define a subset of desired states for the machine and the sequences of those states. This compounded complexity of a computer system makes it extremely difficult to fully understand, analyze, and adequately test to ensure that the software cannot create a situation in which an accident could result. Therefore, we must assume that faults exist in any software system.

Unfortunately, history has proven software does fail, sometimes with disastrous results. Many situations where software has been a contributing element to an accidental loss of life have been documented [40]. The following brief list provides some examples:

1. A British Midland 737 passenger aircraft crashed killing 47 people when the wrong engine was shut off in response to detection of smoke and vibration,
2. A Harrier military aircraft ejection-seat parachute system accidentally deployed without ejecting the seat and the pilot was killed,
3. Audi 500 automobiles have unexpectedly accelerated during gear shifting–two deaths have been attributed to this fault,
4. An Iraqi Scud missile hit Dhahran barracks killing 28 people due to the failure of the Patriot missile defense–a software flaw related to the drift of a clock prevented real-time tracking of the incoming Scud,
5. A Japanese mechanic was killed by a malfunctioning Kawasaki robot.

Perhaps the most prominent and well-researched example of software-related accidents is the Therac-25 [32]. The Therac-25 is a computer-controlled radiation therapy machine used to treat cancer patients. Between June 1985 and January 1987, six known accidents involving massive radiation overdoses by the Therac-25 occurred. Three people died due to complications related to the radiation overdoses, one person died due to the cancer for which she was receiving treatment, and two were seriously injured. These accidents have been described as the worst series of radiation accidents in the 35-year history of medical accelerators. The Therac-25 accidents are discussed in more detail to provide the reader with an appreciation of how subtle and unanticipated latent interactions in software can result in accidents.

The Therac-25 has two operational modes and a non-operational mode. The two operational modes are electron beam therapy and x-ray therapy. In the electron beam mode, the computer controls the current and an adjustable beam energy from 5 to 25 million electron volts (MeV), while scanning magnets are used to spread the beam to a safe, therapeutic concentration.

In the x-ray mode, only one energy level is available: 25 MeV. A beam flattener, an attenuator, is placed in the path of the beam to reduce the energy level of the beam to produce a reasonable dose treatment. The computer is responsible for positioning a turntable and checking the position of the turntable so that a target, the flattening filter, and an x-ray ion chamber, used to measure the energy level of the beam, interrupt the beam path. Without the flattening filter in place, a huge output dose rate results.

The equipment also supports a non-operational mode, a field light position that allows the operator to see where the beam will strike the patient prior to treatment. For the field light position, the electron beam is not active, a stainless steel mirror is placed in the beam path, and a light simulates the beam.

Two different software coding faults related to the control of these different modes of the equipment were identified as the probable sources of three of the radiation overdose accidents. Of the three accidents, two of them were attributed to problems in the data entry routines that allowed the machine to continue before the full prescription had been entered and acted upon. In setting up the machine for use, the operator entered a variety of parameters, including the type of therapy. However, under certain timing conditions during the data entry, if the operator selected electron beam mode and then later scrolled back up the screen to change to x-ray mode, the high power beam was enabled without the flattening filter being positioned in front of the beam. The result was a severe overdose of radiation being given to the patient.

The other accident was attributed to problems encountered later in the program logic, an unanticipated race condition between the software routine that checked if the machine was set properly for treatment and the operator's command input. This fault allowed the machine to be activated in an unsafe state: high energy beam on in the non-operational mode. This condition was a "failure" of a software interlock due to a register "overflow" that allowed a 24-MeV beam to be turned on with the turntable in the field light position without the target in place and without beam scanning. The result was a concentrated electron beam that was scattered and deflected by the stainless steel mirror in the beam's path, resulting in a severe radiation overdose to the patient.

The investigation of these accidents led to identification of several contributing factors such as management inadequacies, lack of procedures for follow-through on all reported incidents, overconfidence in the software, and deletion of hardware interlocks present in previous designs. In 1987, the manufacturer, Atomic Energy of Canada Limited, submitted a corrective action plan to the US Food and Drug Administration that included multiple independent hardware interlocks integrated into the equipment and twenty-three changes to the software to address the safety issues with the equipment.

Leveson and Turner commented in their technical report that summarized their investigation:

> "Although these particular accidents occurred in software controlling medical devices, the lessons to be learned apply to all types of systems where computers are controlling dangerous devices. In our experience, the same types of mistakes are being made in non-medical systems. We must learn from our mistakes so that they are not repeated." [32]

While only a few examples have been cited due to space limitations, these examples illustrate the potential consequences that can occur, even when developers are trying to maintain high quality and reliability. The lesson to learn from these accidents is that software can behave in an unexpected fashion. Consequently, knowledge and methods for enhancing and assessing software surety must evolve.

The challenge is: If software is to be used in a safety-critical application, how can the considerable risk associated with the software be minimized?

1. The software must be developed in such a way that it is extremely unlikely[1] that its behavior will lead to a catastrophic failure.

2. Convincing evidence that the software is extremely unlikely to cause catastrophes in its operational environment, including the presence of hardware failures, must be provided by the software developers to the assessment authority.

This report briefly reviews current practices and research in software surety. Included are applied methods used to design high-consequence systems, and techniques offering improvements in the dependability of software in safety-critical applications. The state of the practice in applied methods is addressed first, and then the current status of cutting-edge research is investigated.

STATE OF THE PRACTICE IN 1994

This section reports on some current practices in use to develop high-consequence software and does not attempt to define or suggest new approaches. In particular, software safety standards, software processes, hazard identification and analysis, design techniques, and implementation methods are briefly covered. Understanding of the practices used in these areas provides a good basic framework for improving software surety. In conclusion, this section provides a sampling of current software surety practices for high-consequence applications among actual industrial development groups.

Readers versed in the area of high-consequence software development are not likely to encounter new material; however, readers unfamiliar with this area will find an overview of a variety of important topics. Wherever possible, an attempt is made to direct interested readers to more detailed literature for each topic.

Software Safety Standards

A standard is an approved, documented, and available set of criteria used to determine the adequacy of an action or object. As computer systems have proliferated into life-critical applications, the need for standards to ensure the adequacy of the safety properties of a system reliant on software to control, monitor, or display information critical to safety has increased. A survey of software safety standards currently in use or currently under development are provided in a paper by P. V. Bhansali [2]. A table from that paper summarizes those standards and is reproduced here as Table 1.

[1]Extremely unlikely in this context is a failure probability on the order of less than 10^{-6} for a system with a one to 10-hour mission time.

Number	Title	Developer
Int Def Stan 00-55	The Procurement of Safety Critical Software in Defense Equipment- Part 1: Requirements, Part 2: Guidance	UK Ministy of Defense
Int Def Stan 00-56	Hazard Analysis and Safety Classification of the Computer and Programmable Electronic Systems Elements of Defense Equipment	UK Ministry of Defense
ISA-SP84 (Draft)	Programmable Electronic Systems (PES) for Use in Safety Application	Instrument Society of America
Mil-Std-882B Notice1	System Safety Program Requirements	US Department of Defense
P1228 (Draft)	Standard for Software Safety Plans	IEEE Computer Society
RTCA/DO-178A	Software Consideration in Airborne Systems and Equipment Certification	Radio Technical Commission for Aeronautics
SC 65A/WG9 (Secretariat) 122 (Draft)	Software for Computers in the Application of Industrial Safety- Related Systems	International Electro-technical Commission
SEB 6-A	System Safety Engineering in Software Development	Electronics Industries Association
UL 1998 (Draft)	Standard for Safety-Related Software	Underwriters Laboratories
N/A	Reviewer Guidance for Computer-Controlled Medical Devices Undergoing 510(K) Review	US Food and Drug Administration

Table 1
Software Safety-related Standards

These software safety standards have a common general approach but vary considerably in their details. The following general steps are common to these standards:

1. Identify and classify system hazards,
2. Identify system functions allocated to software,

3. Identify hazards related to software,
4. Design software and reduce risks or eliminate hazards associated with software and credible hardware failures,
5. Implement software,
6. Verify that risks are at an acceptable level and no new hazards have been introduced,
7. Maintain a configuration management and quality assurance process throughout the product's life cycle.

Analysis techniques or methodologies needed to accomplish these steps are, in most cases, not prescribed or suggested by the standard. These standards cover "whats" rather than "hows." An exception is Int Def Stan 00-55 (see Table 1) that requires a rigorous mathematical approach (formal methods) for specification and design of safety-critical software.

Software Process

The process used in defining and developing high-consequence software is very important. Several models exist to monitor and/or assess system development. Three models an organization might use are the Capability Maturity Model developed by Watts S. Humphrey, In-Process reviews developed by Sandia National Laboratories, and National Security Agency Security Criteria developed by the National Security Agency (NSA). Each of these are discussed below.

Software Engineering Institute's Capability Maturity Model (SEI/CMM): The Capability Maturity Model (CMM) is a framework for characterizing the capability maturity level of an organization's software processes through eighteen key process areas organized into five levels of increasing maturity. An organization uses the CMM to establish a baseline of their current management and software engineering processes, and identify areas where improvement is most needed.

The five maturity levels represented within this model include (from lowest to highest): initial, repeatable, defined, managed, and optimizing [22]. These levels represent general phases of evolutionary improvement of software organizations. The key process areas within each level allow an organization to chart their growth in improving their software management and engineering capabilities. By improving key process areas in accordance with an organization's business objectives, it is possible to establish a controlled and measured process suitable as a foundation for continuing improvement of the organization's software process maturity. It should be noted that an organization may establish a process improvement strategy that involves key process areas from more than one level. In general, the improvement in the key process areas will be most effective when lower-level key process areas have been adequately addressed. A brief review of the capabilities characterizing the basic differences between each level and illustrating the general progression of process improvement follows.

An organization at the ***Initial*** level experiences unpredictable schedules and costs, and poor control over engineering operations. Successes at this level largely depend on individual effort and heroics.

The ***Repeatable*** level requires adequate and appropriate use of several key process areas to implement software processes that can be repeated reasonable well from project to project. This level includes the key process areas of requirements management, project tracking and oversight, project planning, software configuration management, and software quality assurance. If the organization uses subcontractors to supply part of its software, then an additional key process area, subcontract management, must also be addressed. Each of these key process areas must be addressed if an organization wishes to be able to continually repeat its software successes with any predictability. The organization's project management must be able to control commitments, costs, schedules, and changes, which will make it possible to achieve success on similar applications.

To reach the ***Defined*** level, an organization must maintain a process orientation across projects rather than a product orientation for each individual project as evidenced in the Repeatable level. At the Defined level, the organization has developed written procedures to document their software development process, allowing for orderly planning, management, and process improvement. Further, the organization has enforced the use of these defined procedures. At this level, organizations maintain software engineering process groups, use analysis results from peer reviews across projects to derive process improvements, focus on standards and procedures, and initiate appropriate training for their staff. The organization does not depend solely on individual key performers and can transfer their proprietary knowledge and experiences to new employees.

The fourth level, generally referred to as the ***Managed*** level, cannot be reached until the organization's processes are sufficiently well defined and standardized to permit an understanding of processes through measurement and statistical control. At this level, the organization is able to measure and analyze various aspects of its defined software process. In addition, the organization is concerned with product quality management, a key process area that cannot be adequately addressed until the organization has a defined process in place from which the product quality can be monitored.

Finally at the ***Optimizing*** level, analysis and statistical control of processes provide feedback for continuous process improvement during each project so as to optimize the effectiveness of process improvement to product quality. As software technology progresses, the organization's processes will continually change and thus never reach full optimization. Consequently, this level focuses on continual process improvement and defect prevention. An overview of this model and key process areas related to each level is shown in Table 2.

LEVEL	CHARACTERISTIC	KEY PROCESS AREA FOCUS
Optimizing	Management based on automated feedback to achieve process improvement within a project's life cycle, and thus optimize project results.	Process Change Management Technology Change Management Defect Prevention
Managed	Management through measurement orientation where management is by measurement of processes from project results.	SW Quality Management Quantitative Process Management
Defined	Process orientation, where the process group is characterized and initiates process improvement based on results across multiple organization projects. Product engineering processes are documented.	Peer Reviews Intergroup Coordination SW Product Engineering Integrated SW Management Training Program Organization Process Definition Organization Process Focus
Repeatable	Project orientation where the Key Process Areas at this level can be repeated from project to project with some consistency.	SW Configuration Management SW Quality Assurance SW Subcontract Management SW Project Tracking and Oversight SW Project Planning Requirements Management
Initial	People orientation leading to project dependencies on individuals, and unpredictable and poorly controlled management and engineering activities.	

Table 2
SEI Capability Maturity Model Process Areas By Level

In-Process Reviews

In-Process reviews play a major role in the current development of War Reserve (WR - stockpile of weapons) software at Sandia. The Process Guidelines for Sandia WR Software Development [3] calls for numerous reviews including Weapon System / Component Reviews, Software Plans Review, Software Requirements Review, Software Design Review, Critical Design Review, Code Inspection, and Software Test Plan Review.

The general review process is discussed in [43]. This general review process consists of formal and informal reviews. In one use, the term "formal" applies to those reviews that are contractually (or by organization policy) required to be held, and usually are of a management focus performed at major milestone achievements. In another use, the term "formal" applies to those reviews that are performed using rigorous process steps with well-defined entry criteria, inputs, process steps, outputs, exit criteria, and participant responsibilities.

The term "informal" applies to a wide variety of reviews that are not "formal" in the sense described above. There is not a consensus on precisely where the delineation is between the designation of formal and informal. Reviews can be at the management or the technical level. At the management level project status, schedule, resource allocation and technical overview information is typically presented. At the technical level, reviews are typically performed by a small team and are much more focussed and usually more detailed than the management reviews.

Software inspection (Formal In-process Review) is an instance of a formal technical review technique. The reference [7] contains training materials on software inspections. Data presented in this course from published literature indicates that software inspections can detect 50-90% of all defects, lead to a development cost improvement of 10-25%, and reduce overall staff hours for a product by 10-40%. The front-end cost of the product is higher because of the extra time spent in inspections, but reduction in later stage rework (e.g., test rework) results in a lower overall project cost. In addition, the overall long term corrective maintenance costs are reduced. A recent addition to this course is inspection and test data from a large, high-consequence Sandia project that essentially confirms most of the data in the published literature. For high-consequence systems, these front-end inspections are even more important because the costs of operational system failures are substantially higher.

Problems are not solved during software inspections. Any defects found are recorded and the source of the defect resolved later. All defects are corrected before the software is qualified for use. The software inspection checklists given in Table 3 [7] indicate some types of defects that may be identified in each stage of inspection.

Type of Inspection	Sample Checklist for Inspection
Software Requirements Specification (SRS)	-- Are requirement statements open to interpretation? -- Are all resources identified and do they conflict with requirements? -- Are all system initialization actions and values specified? -- Are shutdown or failure actions specified? -- Are invalid data conditions handled? -- Are the requirements testable? -- Are all requirements numbered? -- Are the requirements limited to "what" and not "how"?
Software Design Document (SDD)	-- Are all of the requirements in the SRS satisfied? -- Are interfaces between design entities well defined? -- Are all valid and invalid domains specified for data items? -- Are timing and behavior state considerations adequately addressed? -- Are the design descriptions open to interpretation? -- Does design specify structure of the software, how the data are used, and how control is exercised?
Code	-- Does the source code agree with the SDD? -- Are module parameter lists consistent? -- Are invalid inputs appropriately handled? -- Are there any logic defects and is the logic easily understandable? -- Are comments informative and accurate? -- Does the code conform to coding standards? -- Are modules functionally cohesive?
Software Test Plan	-- Is each SRS requirement addressed by one or more test description? -- Is it clearly stated which features will and will not be tested? -- Is there a pass fail criteria for each test? -- Is the testing environment specified? -- Do the test cases include negative tests?

Table 3
Sample Checklists for Inspections

National Security Agency Security Criteria

The National Security Agency (NSA) developed standards/guidebooks for Trusted Computer System Security Evaluation Criteria, [9] and Trusted Network Interpretation, [39]. The first document, "Orange Book," for computer system security evaluation was also adopted as a Department of Defense standard in December 1985. The National Institute of Standards and Technology (NIST) and NSA have more recently been involved in a joint project to develop a new Federal Criteria for Trusted Systems Technology. The eventual result will be Federal Information Processing Standards. Although there may have been some evolution of the NSA documents over the past few years, the essence of the information and its thrust remains the same.

The concept promoted by these documents is similar to the Software Engineering Institute's CMM. There are four hierarchical "security maturity" divisions that apply to processes used to develop security features provided by the systems. Across these divisions/levels there are four major security categories of interest: security policy; accountability; assurance; and documentation. Within each security maturity division/level there are Security Key Process Areas for each of the four categories of interest. The security maturity divisions are discussed below in more detail.

The trusted computer system evaluation criteria classifies systems into four broad hierarchical divisions of enhanced security protection. The criteria provide a basis for evaluating effectiveness of security controls built into computer system products. The scope of these criteria is to be applied to the set of components comprising a trusted system, and is not necessarily meant to be applied to each system component individually. Hence, some components of a system may be completely untrusted, while others may be individually evaluated to a lower or higher evaluation class than the trusted product considered as a whole system. The second standard cited above, the Trusted Network Interpretation, provides criteria indicating how to interpret the computer system classes of security required when multiple computer systems are connected through a communications medium.

The four broad classes of the trusted computer system evaluation criteria include: Division D: Minimal Protection; Division C: Discretionary Protection; Division B: Mandatory Protection; and, Division A: Verified Protection. A short description of each division follows.

<u>Division D: Minimal Protection.</u> Systems in this class do not satisfy the requirements for a higher evaluation class.

<u>Division C (levels C1, C2): Discretionary Protection.</u> Systems in this class provide for discretionary (need-to-know) protection and, through the inclusion of audit capabilities, for accountability of subjects and the actions they initiate.

<u>Division B (levels B1, B2, B3): Mandatory Protection.</u> Systems in this class have the notion of a Trusted Computer Base that preserves the integrity of sensitivity labels and uses them to enforce a set of mandatory access control rules. Evidence must be provided to demonstrate that the reference monitor concept has been implemented.

<u>Division A (level A1): Verified Protection.</u> Systems in this class are characterized by the use of formal security verification methods to assure that the mandatory and discretionary security controls employed in the system can effectively protect classified or other sensitive information stored or processed by the system. Extensive documentation is required to demonstrate that the Trusted Computer Base meets the security requirements in all aspects of design, development, and implementation.

Examples of "Key Process Areas" that cross these four broad classes include security policy (discretionary access control, object reuse, labels, and mandatory access control); accountability (identification and authentication, audit, and trusted path); assurance (architecture, system integrity, security testing, design specification and verification, and covert channel analysis); and, documentation (security features user's guide, trusted facility manual, test documentation, and design documentation).

The processes used to develop and support software that is trusted become more formal as the class of security "matures" from level D to level A. A "more mature" class of security is required as the classification level of information and other resources protected by the computer system and its software increases, and/or the authorization level decreases for personnel or programs from which the computer system is to provide protection.

The concept of isolation and limitation of security-specific aspects to a relatively small number of components is inherent through the use of a Trusted Computer Base (TCB) to represent all the security-critical components of the system. Therefore, the first principal technique of security function and feature isolation in a small software component is basic to security-critical software applications. The required security level dictates which specific development methods are required to ensure security features are adequate.

Systems at the upper level of Class B and Class A must have verification that the design and its implementation are correct. Formal requirements, language specifications, and finite-state machines are useful techniques for the upper classes. Some of the analysis methods that are currently used to provide security assurance include security fault tree analysis; security failure modes and effects analysis; vulnerability analysis; cryptographic analysis; and covert channel analysis The testing techniques required also vary from basic security flaw hypothesis testing and obvious penetration testing for Class C systems to use of an independent security test team with extensive (over three months) "hands on" formal verification and validation for Class A systems.

Generally, the techniques used for systems that require a lower (C1, C2) and medium (B1) class of security constitute the current best practices. The techniques described elsewhere in this section can generally be adapted for use to provide assurance of security features. The techniques for systems that require a higher (B2, B3, A1) class of security include the all the best practices and some of the more promising research methods, such as formal specifications and finite state machines applied to the computational security kernel of the TCB.

Hazard Identification and Analysis

While utilizing an appropriate software process is important, any practitioner involved in producing high-consequence software must also pay attention to methods and techniques for hazard identification and analysis. In addition, the practitioner must identify which types of safety analysis are pertinent to a particular system. Four basic types of safety analyses may be performed on a system. The four analyses types are called Preliminary Hazard Analysis, System Hazard Analysis, Subsystem Hazard Analysis, and Operating and Support Hazard Analysis [30]. There are many techniques available to perform these types of safety analyses; several are identified below after briefly explaining the different purposes of the cited basic types.

The purpose of the Preliminary Hazard Analysis (PHA) is to identify and prioritize system hazards and to determine safety design criteria and requirements [30]. The system specifications are then written to explicitly address the concerns identified and the design, implementation, and testing are performed with these hazards in mind. As with error detection, identifying safety-related concerns early in the development cycle is desirable to minimize the costs of addressing these concerns. Many times, simple design changes can greatly increase the safety of the system.

The PHA team should include an expert in the domain of the application. In addition, designers of previous systems should be a part of the team if the system is a replacement for an old system. This helps to identify new failure modes introduced through design changes. These experts may participate in brainstorming sessions to identify potential hazards, or more formal group consensus techniques such as the Delphi Technique can be employed [41]. The Sandia WR Software Guidelines specify that this function be performed as part of the Software Adversary Analysis. Sandia engineering procedures also specify that a nuclear safety focused Safety Design/Program Review be held for WR projects [3].

The purpose of the System Hazard Analysis is to evaluate the safety of interfaces between subsystems. The goal of a Subsystem Hazard Analysis is to determine how a subsystem could contribute to a system hazard. Software hazard analysis is a type of subsystem hazard analysis. The output of the software hazard analysis is a set of safety constraints to which the software must adhere [30].

Operating and Support Hazard Analysis identifies and evaluates hazards related to personnel interactions with the system. The output of this analysis is identification of the possible hazards including: identification of potentially hazardous requirements; identification of changes in functional or design requirements; identification of requirements for safety equipment; identification of requirements for interactions with hazardous materials; and identification of requirements for safety training (see table 1).

Safety analysis techniques used to conduct these four types of analyses include design reviews, walk-throughs, checklists, hazard and operability studies (HAZOP), fault tree analysis (FTA), and failure modes and effects analysis (FMEA). FTA and FMEA are widely discussed in the software safety literature [13, 17, 29, 30, 41, and others]. Both of these techniques are used at Sandia. The choice of methods usually is dependent on a person's preference between inductive and deductive problem solving.

The following subsubsections address a variety of hazard identification analysis techniques including fault tree analysis; failure mode effects analysis; Petri nets; and independent vulnerability analysis, before concluding with a brief comparision of the cited methods.

Fault Tree Analysis (FTA)

FTA was developed by Bell Telephone Laboratories in the early 1960's and has been used in many applications including missiles, spacecraft, and nuclear power systems [42]. FTA relies on deductive reasoning. The basic approach is to postulate that the system has failed in a way determined to be safety-critical, then try to find out how the failure could be caused. The set of top events in a FTA is the output of the PHA that identifies possible system hazards. Each top event is analyzed to identify all possible causes of the top event. Logic symbols are used to express the relationships necessary for the undesirable event to occur. Any intermediate events are then further analyzed to determine their causes until all events are expressed in terms of primary events. Primary events are events that cannot be caused by other faults. Defining primary events is part of the analysis. For example, a single bit hardware fault may be considered a primary event.

Software Fault Tree Analysis (SFTA) is a variation of this technique that is adapted specifically to software [1]. The events in this analysis are potential scenarios in which the software could lead to or contribute to a top event. Each top event is analyzed to show that the scenario contradicts the program logic. If it does not contradict the software logic, then a potential safety concern has been identified.

In an experimental application of SFTA, a two-day analysis of over 1,250 lines of Intel 8080 assembly language code discovered a failure scenario that could have resulted in the destruction of a spacecraft. Conventional testing performed by an independent group did not detect this problem [42]. This demonstrates an effective use of the method during the implementation phase of the project.

Failure Mode Effects Analysis (FMEA)

FMEA is an inductive approach to analyzing software for safety concerns. This approach postulates faults in the system and then attempts to predict the effect on the system. Like FTA, this approach has been adapted for use on software systems. In this approach, a fault is assumed in a software module. Faulty logic or inputs to the module might be the cause of this fault. This fault is then propagated to the system outputs to determine the effects on the system.

FMEA analysis is performed using a matrix with software failure modes along the vertical axis and output modes along the horizontal axis. The horizontal axis may also include other columns such as probability of detection and corrective action for the failure [17, 41]. The analysis proceeds by considering the failure modes for each component in the system and predicting the outputs resulting from the failure. Critical failures can then be determined and steps can be taken to protect against the hazardous output.

Hughes Aircraft employs hardware and software FMEAs supplemented with analysis techniques which assess operation under normal conditions and simulation of dynamic timing and failure occurrence conditions to verify their embedded, real-time, control systems that are used in automotive and space applications [17].

Petri Nets

Adapted from the analysis of reachability properties of communication systems by Nancy Leveson and Janice Stolzy, Petri nets can be used to analyze the safety properties of a system [28]. A Petri net models a system as a set of places (states) and a set of transitions. Hardware, software, and human components within a system can be modeled using the common mathematical language. Timing information can also be incorporated into the analysis. The dynamic aspects of a Petri Net are demonstrated by execution of the model; movement of tokens between places occur as conditions for transition are satisfied.

In safety analyses, fault and failure states are added to the model and the model is executed to determine if any unsafe states are reachable from the initial state. This analysis allows identification of the prior states necessary to reach an unsafe state. Once these states are known, the design can be modified to prevent or mitigate the effects of reaching an unsafe state.

Leveson states in her paper, "Unfortunately, Petri nets can be difficult to analyze. For general Petri nets, the reachability problem although decidable, has been shown to be exponential time- and space hard". As the complexity of the system being analyzed increases, the Petri net model's complexity increases exponentially.

An example of industrial use of Petri nets is documented later in the section exploring current approaches used by leading practitioners (Samples of Current Practices in Industry).

Independent Verification and Analysis

Software is generally subjected to an Independent Verification and Validation (IV&V) process to ascertain if the code produced meets its requirements. This IV&V process usually consists of formal analyses of requirements and code (verification) and functional tests that demonstrate implementation of the requirements (validation). The WR Software Development Guidelines [3] require Independent Vulnerability Analyses (IVAN) for software developed by Sandia for WR usage. An IVAN attempts to determine if the software performs functions that are not intended (such as subversions) and also attempts to find vulnerabilities that could affect safety or security. Development and production versions of the code are subjected to static and dynamic analyses to uncover any additional functionality. An IVAN provides additional assurance that the code produced for a WR application does not have intentional or accidental defects that degrade security or safety. However, an IVAN does not ensure correct functioning and safety of the software.

Summary of Analyses

FTA and FMEA share several common weaknesses–they are limited in the types of errors that are detected. For example, algorithm problems, typographical errors in the code, and incorrect variable usage are not detected. Another problem is that the effectiveness of the technique is highly dependent on the abilities and intuition of the person or team performing the analysis. The weaknesses listed here demonstrate the need for a layered approach to safety. Code inspections or module and system testing are more likely to detect algorithm problems, typographical errors, and incorrect variable usage, while FTA and FMEA are more effective in finding basic system safety concerns.

The FTA and FMEA analyses are applicable throughout the design process. As with most techniques, the earlier they are applied in the development process, the more effective they are in finding and correcting potential problems. FTA's applied to the code may detect problems, but the cost of fixing the problem at that stage could outweigh the perceived safety gain.

Petri nets are a different approach to analyzing the safety aspects of a system. Petri nets allow timing information to be incorporated into an analysis; timing information is difficult to incorporate into FMEA and FTA. Because of the exponential increase in complexity of modeling a system using Petri nets, they are more applicable to the earlier stages of development such as requirements analysis. They can also be applied effectively to simplified systems.

The primary focus of an IVAN is not on safety, but rather, on detecting hidden or unintended functionality of the software. The specifics of an IVAN vary according to the preceived threats and the implementation of the system.

Data presented in the paper entitled Traditional Software Development's Effect on Safety shows that safety analysis helps to reduce defects in software products and system failures due to software in specific functions [19]. This paper presents a study in which a control group developed software to control lights at an intersection using "normal" software development methods. Normal in this case is defined as: waterfall model, English specifications, structured design, coding in C, 100% statement and branch coverage, boundary-value analysis and testing, and a limited form of Fagan inspections. The developers in the experimental group used the same basic approach to develop the software except they added preliminary hazard analysis and high-level design hazard analysis steps to the process. There were seven control group teams and five experimental group teams. The results of the errors in the software detected through testing are given in Table 4 [19].

Fault Description	Control Groups Total Errors (# of groups making errors)*	Experimental Groups Total Errors (# of groups making errors)*
Improperly initializes traffic lights	7 (7)	3 (3)
Fails to ensure light changes are valid	737 (7)	162 (3)
Fails to ensure minimum delay times	213 (7)	22 (2)
Leaves all elements off for some lights	32 (3)	11 (2)
Leaves two or more light elements on	13 (3)	2 (1)
Fails to delay between element changes	3 (2)	4 (3)
Fails to monitor hazardous conditions	7 (7)	2 (2)
Fails to keep lights in synchronization	2 (2)	0 (0)
Changes wrong elements	1 (1)	0 (0)
Loses track of traffic-light states	23 (5)	14 (1)

* First number is total errors; number in parentheses is number of groups that contributed to the total errors.

Table 4
Defect Data for Traffic Control Light Software.

The table shows an improvement in the experimental group and provides grounds for belief that given equal processes, deliberate analysis of safety-related scenarios can be beneficial.

Design Techniques

In addition to utilizing appropriate software safety standards, choosing and applying a rigorous software process method, and incorporating hazard identification and analysis approaches as part of a system assessment, the practitioner must also focus on the design techniques that are used. There are a wide variety of design techniques and approaches to designing safer software. Reviewing literature on this subject shows that there are about as many approaches used as there are applications. Each individual application requires analysis to determine effective (both in cost and performance) solutions.

Furthermore, numerous engineering techniques are proposed for the total system design including all hardware components (software is only one component of many that affect the safety of the final system). In the paper "Risk Assessment and Management of Safety-Critical, Digital Industrial Controls–Present Practices and Future Challenges", it is suggested that high level controls be implemented in a heirarchical fashion with the simplest and most reliable ones having the highest priority [10]. The hierarchy is as follows:

1. Physical barriers (e.g. mechanical stops);
2. Static mechanical devices (e.g. burst disks);
3. Moving mechanical devices (e.g. relief and check valves);
4. Simple electromechanical system (e.g. limit and pressure switches);
5. Simple analog or discrete digital electronic system (e.g. overspeed circuits);
6. Simple software based device (e.g. programmable logic controller); and
7. Redundancy (e.g. Triple modular redundancy, standby sparing).

> Dunn continues, stating: "With high level controls in place, the probability that unwanted effector motion will occur is now the joint probability of a control system component failing and producing inadvertent motion, and the probability of the high level control failing to stop it. Risk of the worst case scenario occuring is clearly reduced."

Accordingly in this subsection, discussion focuses on a few approaches that may be used in the development of software based systems that complement a top-level system design as just described in [10], including: hardware and software interlocks, self-checks, n-version programming and recovery blocks, watchdog timers, watchdog monitors, and safety kernels.

Hardware Interlocks

As shown in the Therac-25 accidents discussed earlier, sometimes overconfidence in software can lead to problems. The Therac-25 machine replaced hardware interlocks with software and created a single point of failure in the software. Hardware interlocks are often more trusted than software interlocks because their failure modes are more predictable and better characterized than those of software. This is a highly effective and highly recommended approach when practical.

An example of hardware interlocks used in Therac-20 (predecessor of Therac-25) is a number of switch contacts that provide independent confirmation of machine positioning. Switch contacts are often placed in the power supply of the equipment to enforce proper positioning before power can be applied.

Software Interlocks

Software interlocks help ensure that operations occur in the correct sequence or that an event does not occur without prespecified conditions being met. Event sequencing is an example of a software interlock.

Event sequencing can be achieved using a software "passed parameter". Before initiating an event in the sequence, the passed parameter is checked for a specified value, and then the parameter is updated after the event has taken place. One example is to update the parameter on entry and exit from routines to ensure correct routine sequencing in critical code. When using this technique, it is important that mathematical operations be used to update the passed parameter and not just a simple assignment statement (see Figure 1). This prevents inadvertent entry into the middle of a protected sequence from continuing on erroneously into the next critical function.

Correct Parameter Use	Incorrect Parameter Use

routine_a(parameter)
{
   if (parameter != value_a) error();
 	
   perform critical function
 	
/* now set parameter for routine_b */
 	
   parameter = parameter - number_a;
 	
}
	routine_a(parameter)
	{
	   if (parameter != value_a) error();
	
	   perform critical function
	
	/* now set parameter for routine_b */
	
	   parameter = value_b;
	
	}

Figure 1
Passed Parameter Used to Validate Software Execution Sequence

A similar approach to preventing a critical operation from occurring is to keep a parameter that is used as a "key" in the operation. If writing a value to a port turns on a critical piece of hardware, then the key value is always "ANDed" with the value written to the port so that critical bits will not be set if an inadvertent jump to the section of code outputting the data to the port. Jumping to the unsafe code in Figure 2 would result in a hazardous situation where jumping to the safer code could prevent the situation assuming the key has not been set to allow activation of the output. Another approach is to use an operation such as "exclusive or" to modify a pattern stored in memory to create the critical value needed. This way, the critical value is never directly stored in memory and it must be computed in order to be used.

Normal Code	Safer Code

output_value = critical_number;
output(output_value);
	output_value = critical_number & key;
	output(output_value);

Figure 2
Use of Key to Prevent Inadvertent Execution of Critical Code

Self-Checks

Sometimes sanity checks can be inserted into the software to detect faulty conditions. Some common examples include making sure that pointers to input and output buffers are within predefined limits to prevent overwriting critical data by overflowing input buffers and to prevent inadvertently outputting critical data to ports.

Other sanity checks on structures provide an efficient way to offer protection against errant pointers. This can be done by assigning a structure an id value that is checked before using the structure (see Figure 3). Many language compilers would detect if "double_x" were called with a pointer not of type "Safe_one", but the method listed here could easily be implemented in assembly language which is often used in safety-critical embedded applications. Maintaining a checksum on critical data structures is another widespread technique used to detect corrupted data or errant structure pointers.

Structure Declaration	Structure Use

structure Safe_one
{
    int id = SAFE_ID;
    real x;
    real y;
}
	double_x(Safe_one * Safe)
	{
	    if (Safe->id != SAFE_ID) error();
	    Safe->y = Safe->x * 2;
	}

Figure 3
Use of Structure Id to Prevent Use of Errant Pointers

The software interlocks presented here provide effective protection against several classes of defects including buffer overflows, faulty pointers, and some hardware faults. These techniques are easily added to existing software to improve its safety and can be implemented in new designs with a minimum of overhead.

In general, software self-checks should be used carefully. [30] presents a study that shows several weaknesses with using software self-checks. In this study, twenty-four graduate students were tasked to instrument eight different programs. Only nine of the twenty-four students found any defects. Altogether, a total of twelve defects were found, and of those twelve defects, only six were newly discovered–the other six were previously known to exist. And unfortunately while adding the self-checking code, twenty-four additional defects were introduced into the programs.

N-Version Programming and Recovery Blocks

N-Version programming is used to describe the practice of independently developing versions of software that feed their outputs to a voting unit. Assuming fault tolerance is the goal, the voting unit will take the result that is in the majority. If fault detection is sufficient, the voting unit may shutdown the system in a fail-safe manner.

N-Version programming can be applied to multiple fault classes including design faults, compiler faults, language complexity faults, and implementation faults (see Table 5) [20].

Problem	Solution
Design Faults	N-Version programming using different designs for each version.
Compiler Faults	N-Version programming using different compilers for each version but possibly the same source code.
Language Complexity Faults	N-Version programming using different languages for each version.
Implementation Faults	N-Version programming using different programmers.

Table 5
N-Version Programming Uses

Recovery blocks are similar to N-Version programming with the exception that steps are taken to resolve the differences in a voter output rather than accepting the majority answer. There is more information available on N-Version programming and recovery blocks in [1].

There is a much-publicized study that challenges the independence assumption when developing multiple versions of code [25]. In hardware, multiple versions of a design can be characterized and designed to eliminate common failure modes. In software, common failure modes are often present in independently developed programs. This effect on the actual value of this technique should be closely considered before relying on this technique to achieve high levels of safety. Perhaps the more costly approach that requires separate development teams is inefficient, but using multiple compilers on the same source code might provide a cost-effective method of achieving some level of redundancy.

Watchdog Timers

Watchdog timers are a common approach to detecting timing problems and computer malfunctions. Some microprocessors designed for high reliability uses include watchdog timers as part of the processor architecture. A very generic application of a watchdog timer is to include resetting of the timer inside of a control loop. If the hardware hangs or falls out of the control loop, then the watchdog timer expires and interrupts the system operation. Usually this leads to system reset or a safe shutdown of the system.

The timer can also be a state of health indicator to a hardware interlock. The hardware interlock would be tied to a timer and if the software failed to reset the timer, then the hardware would assume a software problem and deploy the interlock automatically.

Watchdog Monitor

Watchdog monitor approaches have been suggested in the literature for developing high-consequence systems if software is included in the system. The watchdog monitor employs separate hardware that tracks the inputs and outputs of the system to detect hazardous faults and then take recovery action once a fault is detected. Monitors can be used to check for the presence of and range values of I/O, timing limits, and other safety requirements.

The key to developing a good watchdog system is to specify adequate protection without making the watchdog as complicated as the original system. Common mode failure problems with the system being monitored must also be avoided. [27] provides a method of designing a watchdog system by using Linked State Machines (LSMs) to create formal specifications of a hardware watchdog. A later subsection of this report contains a discussion of the author's research.

The development of this or other watchdog monitors for high-consequence systems would require significant development and analysis, and the tradeoffs between the added safety and cost must be analyzed. Analyzing these tradeoffs is also complicated by the difficulty in providing numerical estimates of the reliability and safety of software-based systems.

Safety Kernels

This approach deals with isolating the safety critical portions of the software so that a more thorough evaluation and qualification of the critical code can be performed. The challenge here is proving that other portions of the code do not have side effects that can corrupt the safety critical portion of the code. "Firewalls" are needed between the critical and non-critical portions of the system [14].

A firewall can be constructed in several ways. Two approaches are logical and/or physical separations of code. For example, a logical separation of code within memory will allow the system to eliminate or minimize potential interactions between critical and non-critical portions of the code, such as a stack overflow. A physical separation can provide another level of protection. This type of separation can be achieved by placing critical sections of code on physically separate boards requiring specific sequences of interactions to communicate between the sections. For a further example of the use of this method, the paper by Dr. Knight discussed in the research overview section of this report identifies an approach for using a safety kernel to enforce safety rules.

Implementation

A multitude of implementation choices confront the practitioner. Accordingly, these various choices cannot be adequately addressed in this general overview; however, a few points on the choice of languages and configuration management and quality assurance should be identified and emphasized regarding implementation. Those points are briefly addressed next.

Choice of Languages: Practitioners often do not consider the compiler or other tools used in building the software as a potential source of defects; however, that potential source does exist. Consequently, in very high-consequence systems, it is important to verify that the binary code loaded into the system actually implements the higher-level language design that has been qualified for use. At first glance, this appears to be an overwhelming task. How can this task be simplified?

One approach to simplifying the verification of the binary code is to code the critical portions of the software in assembly language. Using this approach simplifies the verification because the assembly of the code is a direct mapping of the source code to machine language. Assemblers from different vendors will produce identical machine code for a given program as long as macros are not used. On the other hand, this approach greatly increases the complexity of the resulting code and thus, must be considered as a tradeoff. Only those portions deemed sufficiently critical for such an approach should be coded in assembly.

This approach is useful for select critical sections because verifying code written in a higher level language is significantly more complicated. In that situation, the machine code generated by various compilers is no longer unique in most cases. However, some languages, such as Ada, are supported by a compiler certification program that is intended to minimize the functional differences due to different compilers.

Configuration Management and Quality Assurance: Any safety critical project should have good configuration management and quality assurance processes in place. Configuration management is important for ensuring that the code actually installed is the code that has been qualified. Processes for approving changes to code need to be in place and enforced.

Quality assurance provides a somewhat independent check to ensure adequacy of the processes used and the products produced. Verification and validation analysis techniques specific to high-consequence risk reduction can be applied by quality assurance personnel to supplement normal engineering techniques. Both configuration management and quality assurance need to be uitilized when producing high-consequence software.

Samples of Current Practices in Industry

Several leading industrial practitioners were interviewed to identify the approaches being used in their software development efforts. A brief description of practices by John Waclo at Westinghouse Nuclear, Stephen Cha at The Aerospace Corp, and Joseph Profetta at Union Switch & Signal follow to give an overview of current practices in industry.

John Waclo–Westinghouse Nuclear: Westinghouse Electric has been in the nuclear reactor control business for a long time, long enough for a process to evolve to the point where they have extreme confidence in their ability to produce safety critical systems. The product under consideration is a reactor shutdown system. It must work when called upon, but yet there must be no false shutdowns because they are very expensive. It is essentially a process monitoring system, checking trends and key measures and tripping when values are exceeded, or trends are going in the wrong direction.

Westinghouse Electric has been building such systems since the late 1970s. In the late 1970s, the systems were all made up out of discrete, analog logic. In the 1980s it was decided to implement a digital concept in a research effort with the Nuclear Regulatory Commission. The first digital product was a Primary Protection System installed at the Sizewell-B (UK) reactor. It went operational in 1989 and was a plug-compatible digital upgrade of the analog system. There are roughly 100,000 source lines of code in these systems, written in the PLM-86 programming language. Their next major effort is developing a totally digital nuclear plant at the Temelin reactor, a Soviet-designed reactor that was never brought on-line. They are retrofitting sensors, instrumentation, and control.

Westinghouse Electric counts on their development process to ensure the safety-critical properties of their software. They do a tremendous amount of reuse. In the Temelin system, 50% of the code is totally reused from other efforts, 32% is modified, and 18% is new. This is opportunistic reuse, but they are now heading towards planned reuse for common functions. They make heavy use of independent software verification and testing. Formalized regression testing is done by others not involved with the development. They do not do formal verification yet. Temelin accomplishes this by:

1. Using different designers;
2. Using Ada instead of PLM;
3. Using CASE tools for the new development; and
4. Using Formal Methods for the new development.

Stephen Cha–Aerospace Corp.: AeSOP is an interactive failure mode analysis tool recently developed at the Aerospace Corporation. When safety critical software is developed, software safety requirements, representing a subset of system safety requirements that have been refined and allocated to software, must be developed. Software safety requirements often dictate recovery activities software must initiate should hazardous events occur in order to restore the system state to a known safe state.

The Petri net formalism has proven to be effective in analyzing system behaviors. Large and complex industrial systems are being analyzed using Petri nets. In 1987, Leveson and Stolzy demonstrated how the Petri net formalism can be used to perform safety analysis. Starting from a state assumed to be hazardous, immediate predecessor states are generated. Each of these states is examined to see whether it is a "critical state." A critical state is one in which the system had multiple paths (e.g., events) where subsequent states include not only a hazardous state but also a safe one. Their analysis technique attempts to find the conditions that led to the occurrence of a hazardous system state.

Current practice is to do Petri-net-based safety analysis manually. This can result in low productivity, and worse, erroneous analysis. AeSOP (Aerospace Safety Oriented Petri Nets) is designed to correct that problem by providing an automated tool supporting safety analysis. It runs on the X windows system and offers a graphical interface.

AeSOP is based on PNUT, a public domain petri net tool. It was ported to the X window system and modified to use the Merlin-Farber model of timing semantics.

This model is well suited to analysis of safety critical software. In the Merlin-Farber model used by Leveson and Stolzy, a transition is assigned a minimum and maximum firing time. Firing rules are such that a transition must be continually enabled for at least the minimum firing time before it may fire. No transition may remain enabled, without firing, for more than the maximum firing time.

AeSOP provides both a graphical forward reachability graph analyzer and a backward reachability graph analyzer. In the forward analyzer, AeSOP specifies both the earliest and latest time at which the system could enter that state. Forward reachability provides the following features: hierarchical decomposition, graphical state marking, and unpredictable event analysis.

In backward analysis, the user specifies the hazardous system state and the analysis proceeds backwards until either the initial state is reached (in which case the system is not safe), or a deadlock results at an intermediate state prior to reaching the initial state (in which case the system is safe from the specified hazard.)

AeSOP allows interactive failure-mode analysis in a user friendly environment. However the model and the hazard specification must be accurate to correctly perform safety analysis. AeSOP appears to be an effective discovery tool. Errors found in the reachability graph may provide suggestions on how the model or design may be altered to achieve a safe system. The Petri-net formalism effectively forces the user to study the detailed and often subtle system behaviors.

Joseph Profetta–Union Switch & Signal: Union Switch & Signal is in the business of building railroad signaling systems. Because of the operating environment, railroad signaling is an extremely tough safety critical system to develop. Germany will certify nuclear safety shutdown systems, but apparently will not certify railroad signaling systems because of the harsh environment in which they operate.

Historically the companies involved in this area have used immutable laws of nature (e.g., gravity) to assure safety. However, for economic and flexibility reasons the industry is making cautious steps towards software. They've put in a big effort to ensure that the software-based system is as safe as the hardware-based system. The three principles they follow are:

Do not move a switch while a train is on it;
Do not allow trains to collide; and
Err towards restriction.

They are helped by the fact that there is a generally safe state "stop the train."

US&S's current software based product is called MicroLok. The reliability techniques used are self-checking and diversity. Since signaling systems are basically sets of Boolean conditions, they are able to make use of De Morgan's Law to achieve diversity. Results are computed from transformed expressions and then voted. Practice indicates this technique works pretty well, as evidenced by the fact they have had over 12 million hours of operation without an unsafe incident.

MicroLok only gives a qualitative feel for the safety of the systems. Since US&S's customers, the railroads, did not care for quantification, this was satisfactory. However, the industry is moving towards quantitative safety requirements. In particular, US&S has a major contract to provide signaling systems for the Los Angeles Green Line, and they have a quantitative requirement of 107 years' mean time between hazardous event. The technique they are using involves building a high-fidelity model of the system, starting with a VLSI Hardware Description Language (VHDL) description of the CPU being used. A higher level model represents the device model of RAM and ROM. Then they drop the actual code into this model and run a virtual machine with real code. This virtual machine is instrumented and set up for fault injection experiments. They run exhaustive inputs and outputs and match at all levels, including timing and state. Once they have thoroughly tested at one level, they move up a level and do it again. This is extremely time consuming, taking over six months of continuous testing. Commercial tools are generally used for this including VHDL and CASE tools from Mentor Graphics.

US&S is now looking at ways to keep the system safe without having to do validations. They are moving towards Ada and are using object-oriented techniques. They expect this to help in maintenance, robustness, and portability as they move to more powerful processors. They use the IEEE Life-Cycle process definition [24], but tailor it to their specific needs. Following a rigorous process definition helps them end up with a safety critical systems.

STATE OF THE RESEARCH IN 1994

This section of the report reviews several research areas–formal methods, the cleanroom process, and software reliability models–that apply to developing high-consequence software. The methods discussed are not widely adopted at this time, but do show promise as possible ways to improve upon the current state of the practice. The approach repeated in this section is to give an overview of the method and refer the interested reader to more detailed literature wherever possible. The section ends by giving a more detailed discussion of several papers to provide a sampling of current research. Works in progress by Knight, Leveson, Knudsen, Dill, Rushby, and McDermid are discussed.

Formal Methods

Formal methods are characterized by the use of mathematical techniques to prove correctness. Formal specifications of requirements allow an analysis of the completeness (to some extent) and the consistency of the requirements. Basically this involves applying discrete math such as predicate calculus and set theory to form specifications that can be analyzed for completeness and consistency. State diagrams and Petri nets are often used to express the behavior of the system. These techniques are currently much more accepted in the European software development community than in the United States. An exception to this is the growing use of the cleanroom technology in the United States [16]. The cleanroom approach is discussed in detail later in this report.

The strength of formal methods is its use in attacking the problems associated with English language specifications. Most defects are inserted at the front end of the life cycle. However, most defects are found near the end of the life cycle. This suggests many of the problems are associated with the specification process. Without well-specified system and software requirements it is especially difficult to analyze the impact of changes to customer requirements–an analysis that is frequently needed throughout the development life cycle.

There are two important roles in the implementation of a safety-critical system: applications (systems) engineers who understand the functions required of the system as well as the attendant hazards and risks, and software engineers who understand the structuring, implementation, and analysis of software. The communication of system requirements by application engineers, and the derivation of software requirements by software engineers, requires careful coordination and a common understanding of the various levels of specification. This communication is typically hindered by the lack of precision and completeness in the specifications.

More precise specifications are needed. This requires something more suitable to the computer science applications than natural language. Formal (or semi-formal) languages are an appropriate choice. Data flow diagrams, Petri nets, Z (pronounced Zed), and Vienna Development Method (VDM) are all examples of such languages. The use of these languages is an attempt to remove the ambiguity and incompleteness inherent in natural language specifications. In the literature, these languages are instances of "formal methods". Formal methods are used to provide a mathematical approach to expressing system and software requirements and proving that the actual implementation satisfies the requirements.

It is important to realize that even if formal methods are used to specify requirements and verify that the design and implementation accurately reflect the requirements, there may still be no assurance of system safety. The correctness of the implementation may not specifically address the safety concerns for the system—that the system does not do what it is not supposed to do. Such "does not do" safety requirements must also be analyzed so as to be correctly specified. Safety-specific techniques such as hazard identification and analysis must be employed to help ensure that the requirements are adequate to meet the system safety needs.

[15] provides a survey of projects which have used formal methods. This study included 12 projects including five commercial projects, three exploratory projects, and four regulatory projects. A summary of the projects is included in the shaded boxes in figure 4. These summaries are directly taken from Volume 1 of the report generated by the study.

A major conclusion of this report was that formal methods are still immature, but are beginning to be used seriously and successfully by industry to design and develop computer systems. The report concluded that the primary uses of formal methods are "re-engineering existing systems; stabilizing system requirements through precise descriptions and analyses; communication between and among various levels of system stakeholders (e.g., system engineers, software engineers, quality assurance personnel, managers at all levels); and as evidence of "best practice" (in a regulatory and standards context).

One of the barriers to wide acceptance of formal methods is the lack of effective tools to help in the process. Many of the companies presented in the Gerhart study had to perform the proofs manually. Some companies are beginning to offer tools in this area. Formal Systems Ltd. of England is developing a toolset called FDR (Failures Divergence Refinement) with a goal of supporting formal verification of systems using various state machine descriptions, process algebras, and programming languages. Other formal development support tools may be available. A complete survey of such tools are not included here, but [6] provides a valuable starting point for persons interested in formal methodology.

Formal methods are not a panacea for safety-critical software. However, given the proper education and training in formal methods, software engineers could realize large improvements in developing dependable, and hopefully safer, software by applying the mathematics of formal methods in some form to help prove the correctness of the logic.

Regulatory Cluster

Darlington: Trip Computer Software (DNGS)

Ontario Hydro and AECL developed computer-controlled shutdown systems for the Darlington Nuclear Generating Station (DNGS). When difficulties arose in obtaining an operating license from he Atomic Energy Control Board of Canada (AECB), the Canadian regulator for nuclear generating stations, formal methods were applied to assure AECB that the software met requirements. The process was one of post-development mathematical analysis of requirements and code using Software Cost Reduction.

The specifications, code and proofs require 25 three inch binders for each of the two shutdown systems. While there is some discrepancy in the various papers on the amount of code for the two shutdown systems, the definitive word was that one of the shutdown systems (SDS1) has about 2500 lines of code.

The use of the Software Cost Reduction approach finally convinced the AECB that the shutdown system was no longer a licensing impediment.

Multinet Gateway System (MGS)

The Multinet Gateway System is an Internet device that provides a protocol-based datagram service for the secure delivery of datagrams between source and destination hosts. This case is our main computer-security application. (TBACS, one of the exploratory cases, also involved computer-security considerations.) MGS went through a significant portion of the US Trusted Computer System Evaluation Criteria process [NCSC85] and achieved a "developmental evaluation." The process included TEMPEST and communications security analysis. Rigorous mathematics and the Gypsy Verification Environment were used to develop and model security functionality.

From the formal methods perspective, 10 pages were needed to describe the security model and a further 80 pages for presenting the Gypsy specification of the MGS. The underlying operating system has about 6,000 lines of code.

SACEM

The product developed is a certified safety-critical railway signalling system which reduced train separation from 2 minutes 30 seconds to 2 minutes, while maintaining safety requirements. The successful deployment of the signalling system removed the need to build a new third railway line. The developers of the signalling system were required to convince the RATP (the Paris rapid transit authority) that the system met safety requirements. Amongst the numerous techniques used to demonstrate system safety (e.g., fault analysis and simulation) were B and Hoare Logic. The system consists of 9,000 lines of verified code and 120,000 hours of formal methods effort. The new system allows for 60,000 passengers to be carried per hour.

Traffic Alert and Collision Avoidance System (TCAS)

The purpose of the TCAS family of instruments is to reduce the risk of midair and near midair collisions between aircraft. TCAS functions separately from the ground-based air traffic control system. The US Federal Aviation Administration has required, under Congressional mandate, that TCAS II be installed on all aircraft by December 31, 1993. The TCAS methodology was developed and used to formally describe two components of TCAS: the Collision Avoidance System (CAS) Logic and the surveillance system.

There were 7,000 lines of pseudo-code to describe the CAS Logic; the formal description is of comparable size. Work on the surveillance system is in progress.

Figure 4
Applications of Formal Methods Among Various Projects [15]

Commercial Cluster

SSADM Toolset

The British firm Praxis plc. developed a Computer-Assisted Systems Engineering toolset to support the use of the CCTA standard development method called the "Structured Systems Analysis and Design Method" (SSADM). In this project, Z was used to develop a formal specification of the toolset infrastructure and resulted in 37,000 lines of Objective C and a specification comprising 350 pages.

Customer Information Control System (CICS)

The Customer Information Control System is a large transaction processing system developed by IBM. A major portion of a recent release was re-engineered using the Z method and tools at IBM Hursley, UK. CICS is approximately 800,000 lines of source code; of the approximately 50,000 lines of new or modified code, 37,000 were completely specified using Z and about 11,000 were partially specified using Z. IBM claims that the use of Z has resulted in reductions in both development cost and error rates.

Cleanroom Software Methodology

To better understand the Cleanroom methodology, we investigated two industrial applications: one at NASA Goddard and the second at the Federal Systems Division of IBM. The Goddard application focused on the attitude ground support for NASA's "International Solar Terrestrial Physics Satellite." The second application, and the prime object of our study of Cleanroom, was the development of a "COBOL Structuring Facility" (COBOL/SF), which converted old COBOL programs to a semantically equivalent "structured programming" form. The COBOL/SF resulted in a product that was 80,000 lines of code and required 70 person-months of effort. The product was important for demonstrating to IBM management the potential of the Cleanroom methodology.

Software Architecture for Oscilloscopes using Z (Tektronix)

Tektronix in Beaverton, Oregon, used Z to develop a reusable software architecture to be shared among a number of new oscilloscope products. Z was used as a mathematical modelling language to explore design ideas. The models were viewed as being "non-executable prototypes." The software architecture consists of 200 KLOC and 30 pages of Z.

INMOS Transputers (INMOS)

In 1985, a small group of designers at INMOS Ltd. in Bristol, England, began exploring the use of formal program specification, transformation and proof techniques in designing microprocessors. INMOS manufactures advanced microprocessors, and their best known product is the Transputer family of 32-bit Very Large Scale Integration circuits with a unique architecture designed for concurrent multiprocessor applications (processor, memory and communication channels are self-contained on each Transputer
chip).

This case consists of three inter-related projects, all of which use formal methods in some aspect of the design or development of components of three generations of the INMOS Transputer:

1. The "floating point" project: the use of Z to specify the IEEE Floating Point Standard which was applied to two successive generations of Transputer (a software implementation, and a hardware implementation).
2. The use of Z and Occam to design a scheduler for the T-800 Transputer.
3. The use of Communicating Sequential Processes and Calculus of Communicating Systems plus a "refinement checker" in the design and verification of a new feature of the T9000 Transputer, the Virtual Channel Processor.

Figure 4 (continued)

Exploratory Cluster

Large Correct Systems (LaCoS)

The Rigorous Approach to Industrial Software Engineering (RAISE) is a large language and toolset evolved from the Vienna Development Methodology (VDM). Following the funding of RAISE during an ESPRIT I (1985-1990) project and the commitment of commercialization by Computer Resources International (in Denmark), an ESPRIT II (1990-1994) project was formed with the dual purpose of (1) improving the industrial potential of RAISE and (2) transferring formal methods into various LaCoS partners. This survey interviewed one partner at some length with a brief interview with a second partner (there are a total of six "consumer" partners in addition to Computer Resources International, the "producer"). The first, Lloyd's Register, is evaluating RAISE (as well as other methods) on a data acquisition and equipment management system for ship engines. The second, Matra Transport, builds railway and other systems (like GEC Alsthom on SACEM) and has primarily been exploring methodology so far. Both partners are building up consultancy in formal methods and assessment. Since LaCoS is an on-going technology transfer project on a large scale, this case will be important to follow for the remaining three years and beyond.

Token-based Access Control System (TBACS)

The National Institute for Standards and Technology (NIST) is the US measurement and testing laboratory, with numerous ongoing projects underlying standards. In this case, one group in Computer Security Technology had been developing a series of prototype smartcards for cryptographic authentication for network access. Another group in Software Engineering is chartered to look at new technology in support of open systems and other commercial standards. In this case, a staff member from the Software Engineering group was interested in experimenting with formal methods and located the smartcard application as a basis. A toolset and approach was chosen that followed the standard process in Trusted System certification, using a theorem prover to verify that a design meets requirements of a security policy model. In particular, TBACS is a smartcard access control system with cryptographic authentication.

Hewlett-Packard Medical Instruments

This case was of a significant scale and importance of product but it is considered exploratory in that the primary objective was technology transfer. The product is a real-time database (called the Analytical Information Base) of patient monitoring information. Using an HP-developed specification language (a variant of VDM), a specification was produced by a formal methods transfer group and a developer. The transfer effort failed because time-to-market was the key feature and formal methods offered little beyond the high quality already achievable by other means that were consistent with the culture of the organization.

Figure 4 (continued)

The Cleanroom Process

The cleanroom process has been described as the first practical attempt to place software development under statistical quality control and to deliver software with a known and certified reliability [12]. The cleanroom process focusses on defect prevention during the development life cycle rather than defect detection during unit level and functional testing of the software. Defect prevention is accomplished with formal software engineering methods and rigorous correctness verification for creating a correct software design. Testing in the cleanroom process is statistically based rather than selective or functionality based and is focused on the total product rather than the parts to allow an accurate measure of the expected field reliability.

In the cleanroom process, software specifications are required to be documented in a formal mathematical notation rather than natural language. The intent is to force stability and completeness of the software specifications as early in the design process as possible. The specification content is broadened to identify the software requirements for each incremental software release and establish the reliability targets for the product.

A rigorous and formal design method based on structured programming theory [33] is recommended. A limited set of primitives are used to capture design logic, define software structure, and organize the software's data. These primitives are used to refine the software requirements and in construction of a software design that can be assessed and verified at each design step.

Correctness, the equivalence between a requirement and the design, is documented through the use of design primitives. Designs are then verified with correctness proofs by the designers and independent inspectors using the functional technique for correctness verification [36].

A statistical approach [11] to functional testing is used in the cleanroom process. This functional testing is driven by probability distributions of input and output domains that are defined against the requirements and track usage in the software's operating environment. This testing is performed not at the unit level, but at the product level. This testing supports statistical inference for reliability prediction and satisfies the critical validation role for traditional functional testing. A comparison between the software's mean time to failure (MTTF) recorded and the target MTTF can be used to identify what and how much corrective action is needed. Dyer states, "While zero defect software in an absolute sense is unattainable, software with an extremely high probability of not experiencing failures is attainable."

Software Reliability Models

Software reliability can be defined as the probability of failure-free operation of a computer program for a specified time in a specified environment [38]. To estimate or predict software reliability, a model that accurately characterizes the behavior of a software program over time is required. A literature survey shows that hundreds of papers devoted to the subject of software reliability models have been published in the last decade with over 40 different software reliability models proposed. Overviews of some of the numerous software reliability models are available in the following papers: [18], [34], [36].

Goel classifies the various software reliability models according to the nature of the failure process studied. Software reliability models are grouped into four main classifications: time between failure models, failure count models, fault seeding models, and input-domain-based models. All models mentioned in this subsection are referenced and discussed in greater detail in the overview papers.

Time between failure models are based on an assumption that the times between successive failures, say the (i-1) failure and the ith failure, follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. A fitted model is developed with estimates of model parameters derived from observed times between failure during execution of a computer program. This fitted model is then used to assess the reliability of that computer program. Another approach is to treat the failure times as realizations of a stochastic process and use an appropriate time-series model to describe the underlying failure process. Examples of models in this classification are: Jelinski and Moranda De-

Eutrophication Model, Schick and Wolverton Model, Goel and Okumoto Imperfect Debugging Model, and Littlewood-Verall Bayesian Model.

Fault count models are based on the number of faults or failures in specified time intervals rather than the times between failures. The failure counts are assumed to follow a known stochastic process with a time dependent discrete or continuous failure rate. Parameters of the model can be estimated from the observed failure counts or failure times. Estimates of the software reliability then can be obtained from the relevant equations from the model. The Goel-Okumoto Nonhomogeneous Poisson Process Model, the Goel Generalized Nonhomogeneous Poisson Process Model, the Musa Execution Time Model, the Shooman Exponential Model, the Generalized Poisson Model, the IBM Binomial and Poisson Models, and the Musa-Okumoto Logarithmic Poisson Execution Time Model are examples of fault count models.

In fault-seeding models, known faults are planted (seeded) in a computer program. The computer program is then tested. After testing, the numbers of exposed seeded and indigenous faults are counted. Using combinatorics and maximum likelihood estimation, the number of indigenous faults in the program and the reliability of the computer program can be estimated. The most popular and basic fault seeding model is the Mills' Hypergeometric Model.

The approach using input-domain-based models is to generate a set of test cases from an input distribution, operational profiles. Difficulty in estimating the input distribution results in partitioning the input domain into a set of equivalence classes. An equivalence class is usually associated with a program path. A reliability measure is calculated from the number of failures observed during symbolic or physical execution of the sampled test cases. Examples of input domain models are the Nelson Model and the Ramamoorthy and Bastani Model.

Essentially all of the software reliability models discussed are reliability growth models. These models are based on a test, find, and fix approach. A computer program is tested until it fails. The failure is investigated and the fault is corrected. The program then undergoes further testing to uncover the next failure. This process repeats until sufficient data is available to predict the reliability of the final "corrected" program based on the observed failure data from the previous versions.

These models have several general underlying assumptions that, if violated, can allow the models to produce reliability estimates that are too optimistic–a bad trait for a reliability model. These assumptions are:

1. As faults are removed from a program, the program exhibits a true reliability growth, fewer failures occur in a specific time interval. This assumption ignores the possibility of new faults being introduced during the fault removal process.

2. All failures are detectable. Some faults may manifest themselves as subtle changes in data that may not be detected immediately. Also, experience shows that some faults may be extremely difficult to find by testing [34]. These faults may manifest themselves as program failures very infrequently. How long will you have to test to find the last fault? The answer to this question is usually to test much longer than you can afford from a cost and time standpoint.

3. The test cases used to generate data to estimate software reliability are representative of the operational usage. If the end user of the software chooses to use the software

functions in different computing environments, proportions or patterns than the test cases used to develop the reliability estimate, a lower reliability may result. Musa stresses the importance of operational profiles that reflect the actual user profiles for generation of test cases [38].

4. The time between program failures are independent. If test cases are chosen at random, one could argue that the time between failures should be independent. However, current practice is to use operational profiles and test high probability user paths. Also, detection of a critical fault during testing tends to focus testing on specific modules of the program. Designers and testers generally refer to these program modules as "design holes" or "bug clusters."

Software reliability growth models are used to predict, from actual failure data collected during testing, the achieved product reliability and the testing effort necessary to achieve a target reliability. The accuracy of these models, as measured by actual versus predicted counts of failure, has varied from one project to the other [4]. The application of software reliability models to situations that require extremely high-reliability, failure probabilities as low as 10^{-9}, is not practical. The test time to demonstrate a low failure probability requires decades of testing. In a paper addressing this topic [5], Butler and Finelli concluded, "At even the most optimistic improvement rates, it is obvious that reliability growth models are impractical for ultra reliable software."

Samples of Current Research

Research in software surety can focus on many different areas. Which of these areas offer the most promising advancements? To provide some insight into the answer of that question, current efforts by a few leading researchers are discussed next. Those efforts include work in progress by Knight, Leveson, Knudsen, Dill, Rushby, and McDermid. Specific references are cited when available.

Dr. John Knight–University of Virginia: Two reports describing current research efforts under investigation by Knight, [26] and [44] are reviewed next.

Safety-Critical computer applications - the role of software engineering: A precise definition of what software safety means is essential before any attempt can be made to achieve it. Informal, intuitive notions of software safety must be rejected if for no other reason than to define the legal responsibility of the software engineer.

In isolation, software is never unsafe. But in practice, software is never used in isolation. Software is merely one of the many components of a system. For the most part, none of the components of the system are unsafe in isolation. Furthermore, nothing about these components (including software) changes when they suddenly become part of a system. It is the system that defines the hazard. There is no such thing as a software hazard. However, as a result of the system design, functional and non-functional requirements are imposed on all of the system's components including the software. For safety-critical systems, these requirements will include various dependability requirements for the software just as for the other components. Thus, the definition of software safety is focused on the software's requirements specification which is derived from formal considerations of system safety.

The paper goes on to define software safety as follows:

Software is safe if it can be shown to comply with its recovery functionality specifications and either comply with its safety-critical functionality or detect execution-time errors in the implementation of the safety-critical functionality and comply with its failure interface specifications instead.

Recovery functionality specification is the additional functionality imposed on the component by the possibility of one or more system components failing. Safety-critical functionality specifications define the functionality that must be provided during normal operation in order to avoid hazards. Failure interface specifications define the functionality that must be provided with a certain probability in the event that the component is unable to comply with its safety-critical functionality specifications.

To clarify the assignment of responsibility, the authors define software safety failure as:

A software safety failure occurs when the software violates its safety specification–that is, it fails to comply with either its recovery functionality specifications or fails to comply with its safety critical functionality and fails to present its failure interface.

Given these definitions it becomes possible to talk about safe software. The achievement of safe software requires that safety specifications be prepared, that the software be carefully implemented, and that the compliance of the software with its safety specifications be verified. The software is defined to be safe to the extent that this verification is successful. A system, of which the software is a part, might not be safe because the safety specifications might be defective in a manner that is unknown to the software engineer.

<u>A safety kernel architecture</u>: A recent report [44] by Kevin Wika and John Knight details the design of a safety kernel architecture. For years the security community has had the notion of a security kernel. The safety kernel enforces safety policies independent of application programs and permits verification of properties of a small kernel rather that large amounts of application software.

The kernel mediates all exchanges between the software and devices under control. It has the following five benefits to a safety critical system: 1) it can ensure enforcement of safety policies, 2) the fact that the kernel is relatively small facilitates both implementation and verification, 3) the application software does not have to worry about safety policy directly, 4) the kernel is in a position to monitor device activities for consistency, and 5) the kernel makes reuse more attractive.

The first problem that has to be addressed in the design of a safety kernel is to determine exactly how it will interact with a large amount of "untrusted" software. The architecture chosen, shown below in Figure 5, has the kernel operating on top of an existing operating system. Thus the kernel must be able to deal with operating system failures. A plausible design for the safety kernel would be to have it run as a user-level server process. This would isolate it from application software failures, and permit it to monitor the application devices independently.

It appears that an operating system failure would leave a kernel designed as above vulnerable. However, most failures beneath the kernel can be dealt with using techniques such as coding to detect data corruption, liveness checks and time-out mechanisms to ensure timely direction of devices, and independent sensors for closed loop control of devices.

The one architectural feature added is the safety kernel monitor which operates on a minimal separate hardware unit and possesses the minimum functionality required to restore the system to a safe state if it does not receive a timely "heartbeat" from the safety kernel.

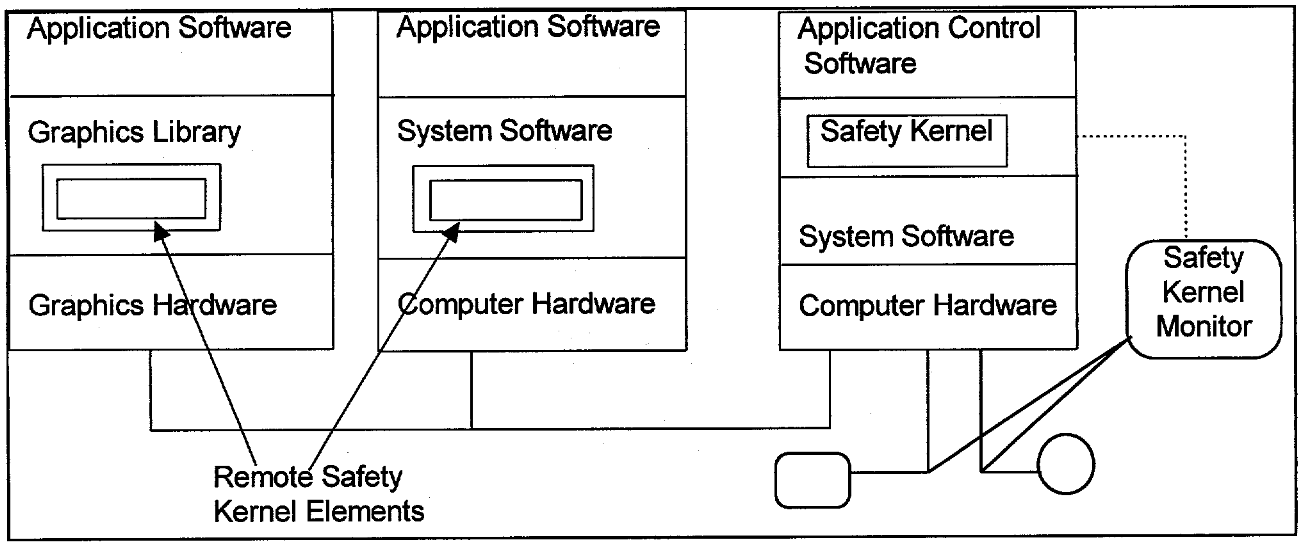

Figure 5
Safety Kernel Architecture

There are three attributes of a safety policy that determine which safety requirements should be implemented in the kernel. These attributes are enforceability, generality characteristics, and functionality. A safety policy that can be enforced by the safety kernel without interaction with the application is referred to as kernel-enforced. An example of a policy which can be kernel enforced is that two devices must not be on at the same time. The opposite of kernel enforced is application enforced. An application enforced policy can be supported by the kernel, but it cannot be wholly enforced by the kernel. As an example consider the policy: "The result of an application-specific algorithm must never yield a result greater than X."

A safety policy is neither inherently kernel enforced nor application enforced. It depends on the characteristics of the policy. The determination of which policies will be kernel enforced is based on maximizing the benefits of a kernel architecture. For example, enforcing a particular policy might provide support for software safety, but result in undue loss of generality and increased complextity of verification. The paper gives some heuristics for making this determination.

The functionality of a safety policy depends on the kinds of things that have to be done to make a system safe. There is no formal notion of safety that has the same degree of rigor as that found in the statement of security. Thus, a mathematical attack is precluded. To permit analysis the functionality is divided into four areas: control of peripheral devices, application software activity, device failure detection, and response to failure.

When controlling peripheral devices one needs to check safe, interlocks, parameter values, etc. to make sure that the device remains in a safe state. A policy affecting application software activity might be to require that a parameter check be done before an output value is sent to a device. Failure detection falls to a component of the kernel that has the exclusive task of periodically sensing the state of devices and comparing the state to the expected state. The response to a failure might be "if the state doesn't match, deactivate the X-Ray unit."

As may be apparent from the above discussion, although a safety kernel is a general architecture, it will have to be instantiated for any particular application. The issues with configuration of the kernel are how to facilitate verification of an instance of the kernel and how to best promote its reuse by a range of applications.

The feasibility of the above research is being evaluated in the context of a Magnetic Stereotaxi system being developed at the University of Virginia. This is a device for performing neurosurgery.

Nancy Leveson–University of Washington: Leveson, Heimdahl, Hildreth and Reese [31] have developed and validated an approach to writing requirements specifications for process-control systems as well as a specification language which supports the approach. The approach is based on Harel's State Charts [21] with significant modification for usability. The method, called Requirements State Machine Language (RSML), was validated by formally specifying an industrial aircraft collision avoidance system (TCAS II).

The technique described is important for a number of reasons. The first reason is that it has been successfully used to write formal requirements for a real, complex, process-control system. Few examples exit of the application of formal methods to complex, real-world, system requirements specification. The second is that, although formal, the specification language is readable and reviewable by application experts who are neither computer scientists nor mathematicians.

The technique involves building a black-box state model of the system (not just the software or even just the computer) describing the behavior of the components and their interface. No additional information is included in the model. The black-box model separates the specification of requirements from design, simplifying the model and making the requirements model easier to construct, review , and formally analyze. This includes the ability to verify consistency of the control model with the system goals and constraints, the generation of standard engineering and system safety analyses such as fault trees, and the application of formal correctness and robustness criteria to the specification model.

Figure 6 shows an example of a state diagram. It is taken from [31]. There are two states,

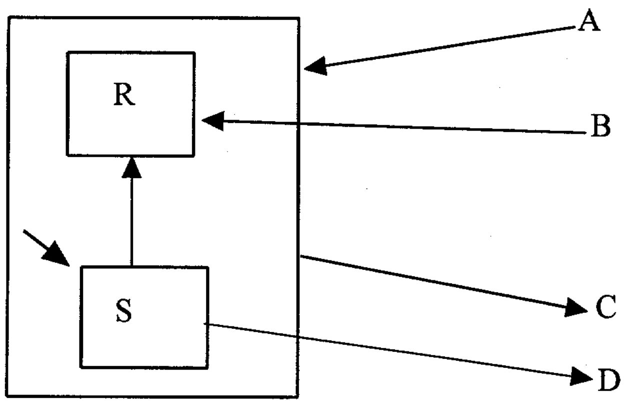

Figure 6
Simple State Diagram

R and S. The default state is S (as indicated by an arrow that has no source). There is a transition from S to R. In this example, the two states R and S are grouped into a superstate. A transition that ends at a superstates border (e.g. A) leads to the (required) default state within the superstate. There may also be transitions to specific states within the superstate (e.g. B). Transitions out of the superstate may originate from the border or from an inner state (e.g. C and D respectively).

Each state machine in RSML may be divided into three parts: The input variables, the output variables and the state machine (figure 7).

The triggering event for state transitions must also be defined. An early attempt to write these conditions used pure predicate calculus. External reviewers did not find it natural or reviewable, and indeed when another, simpler to review, notation was developed, logical errors in the predicate calculus specification were uncovered. Instead of logical phrases, a tabular representation of disjunctive, normal form, called AND/OR tables were used.

The far left column of the table in figure 8 lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements are true. The above table is equivalent to:

$$((\text{Expression-1} \wedge \text{Expression-2}) \vee (\text{Expression-1} \wedge \text{Expression-3})).$$

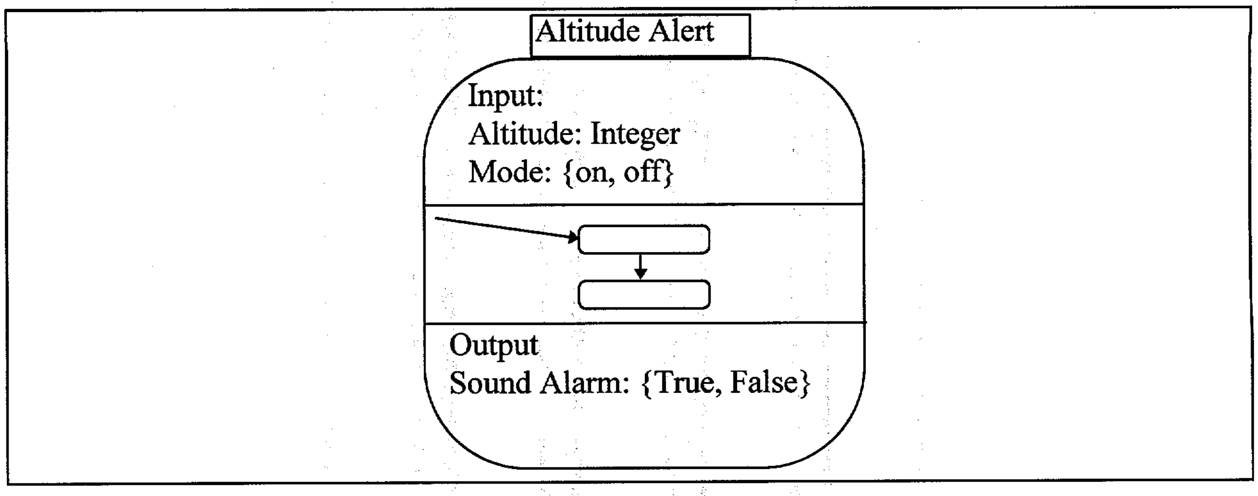

Figure 7
State Machine

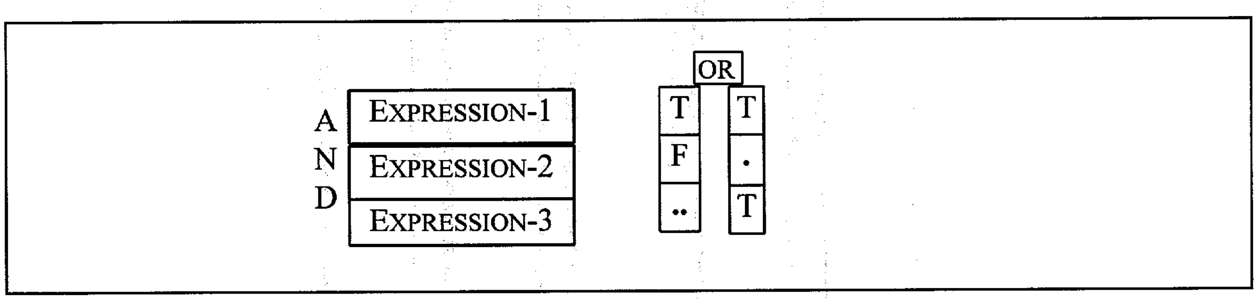

Figure 8
AND/OR Table

The don't cares (".") in the table help to make omissions become more apparent to the application experts. Typically they also add an English language description of the guarding conditions on each transition.

The above is just a summary of some of the aspects of RSML. The paper provides complete details [31]. This appears to be an important method for formally specifying requirements for safety-critical systems, at least in the arena of process-control.

Knudsen–University of New Mexico: Knudsen proposes a method of using Linked State Machines (LSM) to specify and design a watchdog system for a Weapon Control Unit (WCU) [27]. The watchdog is realized as an application specific integrated circuit (ASIC). The WCU without the watchdog system is shown in Figure 9 [8], the WCU system including the watchdog ASIC is shown in Figure 10 [8]. As shown in the diagram, the watchdog ASIC contains two functional blocks, a Watchdog and a Supervisor/Control Unit.

The watchdog unit monitors external WCU signals and provides active feedback of the system status to the Supervisor/Control Unit. The external signals were chosen based on two criteria: 1) was the signal safety critical, and 2) was the signal instrumental in indicating the state of the WCU. Eight signals were chosen for monitoring. Seven were deemed to be safety critical, and one was included to provide WCU state information.

The goal of the watchdog design was to achieve higher safety levels without greatly reducing the reliability of the system. To achieve this, accurate specification of the watchdog must be available, and the description must be consistent with the operation of the WCU. The approach taken in this paper was to develop the watchdog specification using LSMs. LSMs were chosen for three reasons: 1) the watchdog specification operation depended on event orderings that lended themselves to state machine representation, 2) the ASIC could be designed using state machines, and 3) equivalence-based analysis of LSM is a more tractable problem than state reachability analysis [8].

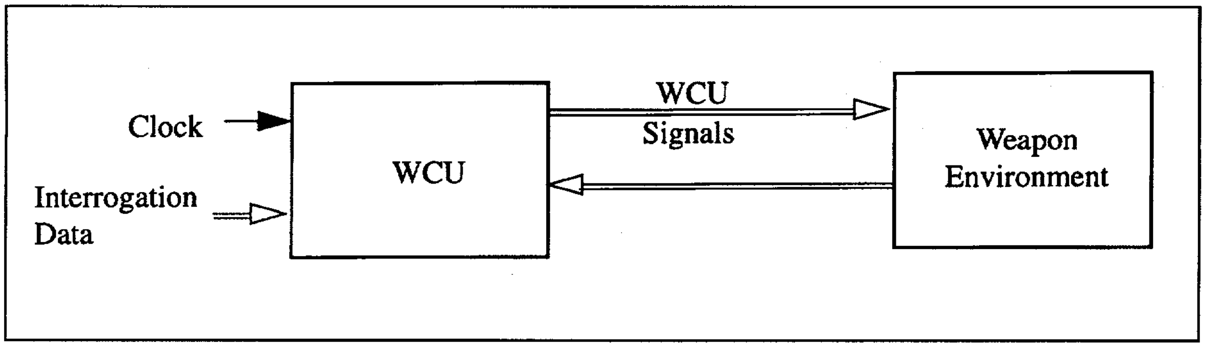

Figure 9
WCU System Without Watchdog

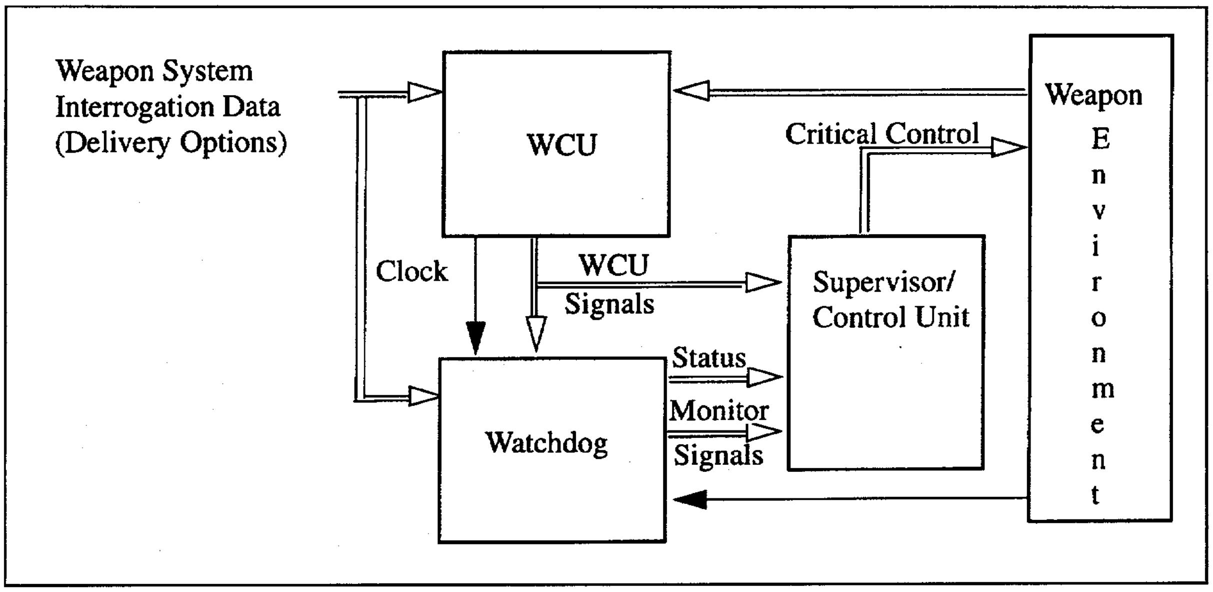

Figure 10
WCU System With Watchdog

An LSM consists of a collection of coupled state machines, an observer, and an initial state set. The coupled state machines are specified using a restricted form of petri nets. LSM behavior is governed by petri net transition rules where states are altered by transition firings.

The LSM is used to model a system and some part of its environment. The dependencies of the system on its environment are characterized by the LSM's initial state set.

Using this LSM notation, a formal description of the watchdog function was developed. In addition to developing the watchdog LSM, a LSM for the WCU control signals was developed to describe the operation of the system to be monitored. After creating these LSMs, equivalence class analysis is performed on the LSMs to verify that error signals will be sent to the Supervisor/Control unit only if the WCU exhibits "out of specification" behavior.

After validating the watchdog LSM design, the LSM can serve as a basis for design of the watchdog ASIC.

The design of the Supervisor/Control Unit was not covered in either [8] or [28]. The intent of the Supervisor/Control Unit is to interpret the status signals received from the watchdog and take action based on the status of the system. The "failsafe" operation of the system would be configurable by the design of this unit.

To summarize, the approach was to use LSMs to formally define the system to be monitored. The LSM representation was useful both in the validation of the watchdog design and in the realization of the system hardware in the form of an ASIC. As with any watchdog monitor system, the tradeoff in this approach is between the increased safety versus the decreased reliability of the system. An attempt to maximize this tradeoff was made by using a formal specification to help ensure compatible operation of the watchdog design and system design. Also, the formal specification was used in the design of the watchdog ASIC so the chance of an defect in the ASIC state machine was minimized. By using an ASIC, minimal hardware is needed to implement the watchdog which also helps to improve the reliability of the design. The conclusion of this report was that this approach is applicable to systems that generate event orderings based on receive data and which are implemented as state machines [8], [28].

David Dill–Stanford University, John Rushby–SRI International: The system developed by Prof. David Dill at Stanford employs a transition-rule-based description language for concurrent systems, called "Murphi". Murphi is a relatively high-level language that provides complete finite-state verification. It has been tested extensively on real-world examples. The system implements basic verification by state enumeration.

Dr. Rushby has developed a system, called PVS, which attempts to combine an expressive specification notation with a significant degree of automation. PVS is able to support a powerful type system containing predicate subtypes, dependent types, and abstract data types. The theorem prover provides a powerful set of interactive proof construction primitives that can be used to debug proofs and specifications efficiently. It has also been tested with real-world examples.

Dill and Rushby are working together to join their two systems to combine the strengths of finite-state and theorem-proving approaches in order to build an environment that is a powerful aid in the design and debugging of complex, high-assurance systems.

Most of the real-world experience with the above systems has been with hardware systems. Dill believes that the tools can be valuable in developing a software system. The primary domain of application is small, tricky algorithms involving concurrency and/or non-determinism. These applications often have subtle bugs (so verification has a high payoff), and can often be analyzed without dealing with too many states (so they are feasible). This would not be solving the problem of "programming in the large"-rather it would be direct to the core code that is most likely to cause reliability problems due to flaky, nonrepeatable errors.

John McDermid–University of York: The effort reported [23] by Hutcheon, Jepson, and McDermid attempted to apply techniques contained in the United Kingdom's Ministry of Defense interim standards IDS 00-56 to the software development process. Specifically, they looked at Ada development tools for developing high integrity software. They discuss the requirements and a generic development process for the production of such software and consider how the tools currently in general use meet the needs. Finally, they identify an ideal toolset for developing high integrity software, and match this set against tools which are currently commercially available.

The context of this work is the development of a safety-critical software component. A significant factor which underlies this work is that software integrity cannot be measured directly, although certain manifestations of lack of integrity may be recognizable. The integrity of software can be increased, decreased or maintained by the development process, although it can never exceed that of the requirements. Tools which perform transformations to carry out steps of the development process can directly affect integrity.

Defects can be made at any stage of the life cycle and can find their way into the software. So the integrity of tools have to be looked at in two different ways: 1) the defects they introduce into the system, and 2) the defects from "upstream" which they are able to filter out. The iterative nature of development complicates this process.

We can make a relatively simple correlation between broad quality and the likelihood of a tool failure, even though we cannot directly relate the tool failure modes to application hazards. A tool which has fewer failures will, in general, introduce fewer faults into a program. However, this link cannot be made to hazards directly as the less trustworthy tool may place its faults in unimportant areas than the more trustworthy one.

The idealized software development process is shown in Figure 11. The deliverables produced by the development process can be separated into three groups according to "closeness" to the software which is installed in the target system:

1. The executable system;
2. The requirements, specification, design, source code, object code; and
3. All verification and validation material.

The development process for the British Aerospace Eurofighter software is shown in Figure 12. It was analyzed to get a feeling for the current state of the practice in the production of high integrity systems. The toolset used is rather sparse when compared to the idealized development process of Figure 11.

How steps from the idealized process shown in Figure 11 are actually utilized in the research laboratory, leading edge industrial development centers, and in general is shown in Table 6.

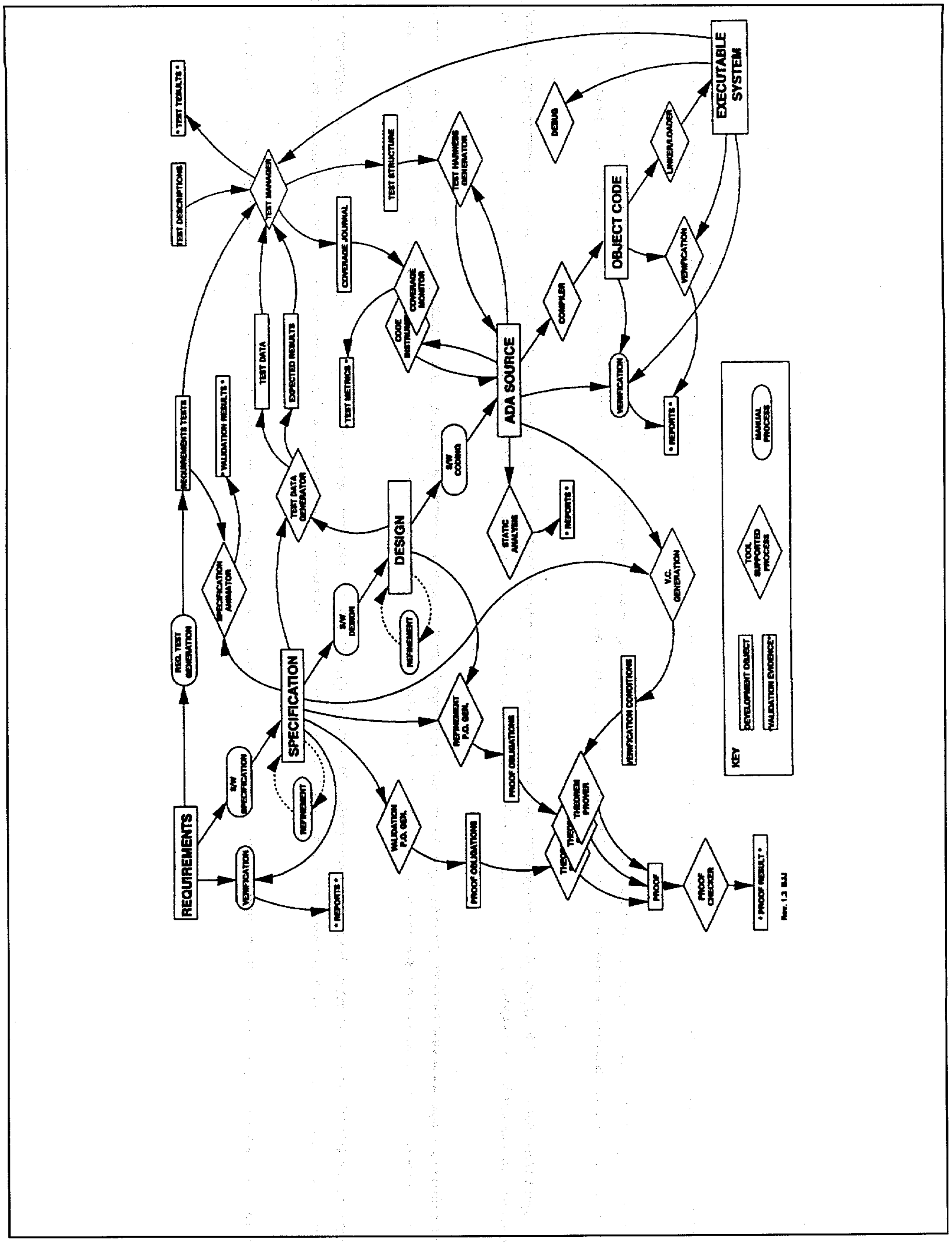

Figure 11
The Software Development Process

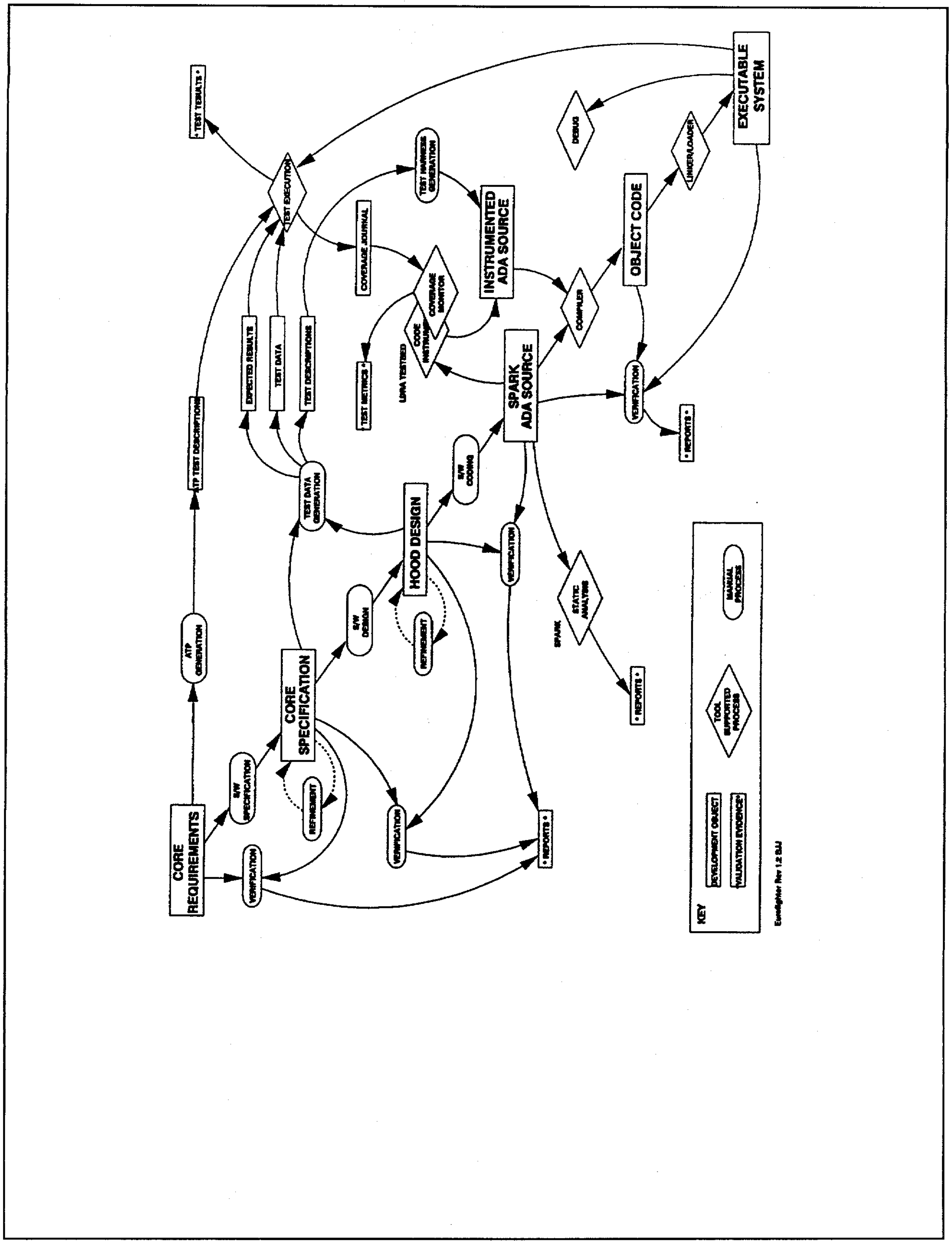

Figure 12
BAe/Eurofighter Development Process

Activity	Area of Use*		
	Research Labs	State of Industrial Art	Widespread
formal software specification	MA/TS	MA	NP
formal software refinement	MA/TS/ME	MA/NP	NP
formal specification animation	TS/ME	MA	NP
requirements/specification verif.	MA/TS	MA	MA
validation proof obligation gen.	MA	NP	NP
theorem proving	TS/ME	TS	NP
proof checking	TS/ME	MA/TS	NP
software design	MA/TS	MA	MA
formal design refinement	MA/TS	MA	NP
refinement proof obligation gen.	MA/TS	NP/MA	NP
software coding	MA/TS	MA	MA
static analysis	TS/ME	TS/ME	NP
timing analysis	TS/ME	MA/TS	UC
resource analysis	IF	MA/TS	UC
verification condition generation	TS/ME	MA/TS/ME	NP
test data generation	TS/ME	MA/TS	NP
test management	NP	TS	TS
code instrumenting	TS/ME	TS/ME	IF
test coverage monitoring	TS/ME	TS/ME	IF
test harness generation	TS	MA/TS	MA
debugging	TS	TS	TS
compilation	ME	ME	ME
link/loading	NP	ME	ME
load verification	MP	TS	UC
source/object verification	TS/ME	MA/TS	UC
source/executable verification	TS/ME	MA/TS	UC
requirements testing	NP	MA/TS	MA
infrastructure	TS	TS	TS
process coordinator	NP	MA/TS	UC

***NP** - Not practiced, or not an active area of study; **UC** -Uncontrolled; **IF** -Informal; **MA** - Manual; **TS** -Tool supported; **ME** -Mechanical or automated.

Table 6
Industrial Usage of Software Development Steps

Although tools which are currently commercially available cover only parts of the idealized development process, the technology to support almost the whole process exists in experimental form and could be assembled in the medium term (ignoring integrity level requirements).

Summary and Conclusions

Software fails. The consequence of software failure has, in some cases, been the loss of life. A primary challenge for designers of high-consequence systems is assuring that the software component of the system does not cause or allow an unsafe system state.

A sampling of current research and practices for software surety in high-consequence system applications has been provided. This sampling supports the following conclusions.

State of Practice

The state of the practice is capable of producing software that satisfies reasonably low software reliability concerns. Application of standards, good software engineering methods, software inspections, and process improvement methods provide assurance that the software has been designed for reliability. Use of reliability growth models and measures of test and operational failures provides reliability quantification methods satisfactory for low reliability concerns.

For certain application domains with small, contained, critical components some technologies are available that, in combination with a well-defined engineering process, remove many variables that impact software reliability. Such technology methods include fault tree analysis, failure modes and effects analysis, petri nets, and watchdog methods. Although software reliability quantification at this level is difficult to verify, failure and defect measurement are valuable indicators. Quantifying the risk associated with the use of software in high-consequence applications requiring failure probabilities lower than 10^{-9} is currently impractical. Software reliability growth models can be most effectively used in applications where the failure probabilities are not less than 10^{-3}.

State of Research

The state of the research is attempting to develop techniques and methods that are capable of producing and/or verifying the reliability of software for high consequence applications. There are isolated instances of projects that have applied some of the available research methods, but without the capability to quantify (i.e., validate through measurement) the level of reliability. Formal methods have been applied to very small, critical components. The cleanroom process combines some of the best aspects of good software engineering processes with specific analysis techniques to ensure correctness. The research efforts primarily amplify the first principles of isolation, inoperability (no single point of failure), and incompatibility (instantiation of high-consequence operation cannot be accomplished without some unique signal compatibility/verification).

Best Technologies

Two techniques appear to offer the most promise in assuring the safety of a system that contains software. These techniques are formal methods for software development and independent watchdogs. The growing consensus among researchers and system developers is that rigorous mathematical (formal) software design methods augmented with system safety analyses provide improvement in the dependability of software in a safety-critical application. Formal design methods for capturing requirements have been successfully applied to several software development projects. Formal method tools such as Z language and toolset, VDM, are currently available. However, at this time, formal methods are not in widespread use by industry.

Simple independent watchdogs that monitor the operational states of software-based systems and have the capability to restore a system to a safe state, have been proposed by several researchers. This concept has progressed as far as the design stage of a weapon control unit for a Sandia National Laboratories development program.

It should be noted that formal methods and independent watchdogs are not "Silver Bullet" solutions that guarantee absolute safety. However, these techniques when applied appear to offer an improvement in the safety of a system. A strong need still exists for more research into methods and tools to assess software surety.

REFERENCES

[1] T. Anderson, Resilient Computing Systems, Vol. 1, Wiley-Interscience, New York, NY, 1985.

[2] P. V. Bhansali, "Survey of Software Safety Standards Show Diversity", in Computer, pp. 88-89, January 1993.

[3] M. A. Blackledge, Process Guidelines for Sandia WR Software Development, Sandia National Laboratories, July 1992.

[4] S. Brocklehurst, P. Y. Chan, B. Littlewood, and J. Snell, "Recalibrating Software Reliability Models", IEEE Trans. on Software Engineering, Vol. 16, No. 4, April 1990.

[5] R. W. Butler and G. B. Finelli, "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability", Proc. of the ACM SIGSOFT91 Conference on Software for Critical Systems, pp. 66-7, Dec. 1991.

[6] D. Craigen, S. Gerhart, and T. Ralston, An International Survey of Industrial Applications of Formal Methods, Volume 1 Purpose, Approach, Analysis, and Conclusions, Naval Research Lab, Washington, DC., September 1993.

[7] INTEC CS776, Software Inspections (Formal In-Process Reviews) Course Notes, Sandia National Laboratories September 1992.

[8] L. J. Dalton and H. K. Knudsen, A Failure-Resistant Architecture and Methodology for Reliable Specification and Design presented at High Consequence Operations Safety Symposium, Unpublished, Sandia National Laboratories, July 1994.

[9] Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense, Fort George G. Meade, MD., December 1985.

[10] W. R. Dunn, et al, "Risk Assessment and Management of Safety-Critical Digital Industrial Controls - Present Practices and Future Challenges." University of Southern Colorado, 1994.

[11] M. Dyer, "Statistical Testing: Theory and Practice." Tutorial at 7th International Software Testing Conference, 1990

[12] M. Dyer, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, Inc., New York, NY, 1992.

[13] P. Fenelon and J. A. McDermid, "An Integrated Tool Set for Software Safety Analysis", J. Systems Software, pp. 279-290, 1993.

[14] K. B. Gallagher and J. R. Lyle, "Software Safety and Program Slicing," The Proceedings of the Eighth Annual Conference on Computer Assurance, 1993.

[15] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems", IEEE Software , Vol. 11, No. 1, p. 21-8, January 1994.

[16] W. W. Gibbs, "Software's Chronic Crisis" Scientific American, pp. 86-95, Sept. 1994.

[17] P. L. Goddard, "Validating The Safety of Embedded Real-Time Control Systems Using FMEA", The 1993 Proceedings of the Annual Reliability and Maintainability Symposium, pp. 227-230, 1993.

[18] A. L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability", IEEE Transactions on Software Engineering, Vol. 11, pp. 1411-1423, 1985

[19] L. D. Gowen and M. Y. Yap, "Traditional Software Development's Effects on Safety" Proceeding of the 6th Annual IEEE Symposium on Computer-Based Medical Systems, pp. 58-63, 1993.

[20] L. D. Gowen, "Developing and Analyzing High-Level Designs for Safety-Critical Software Systems", Conference Proceedings - IEEE SOUTHEASTCON 1993, 1993.

[21] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", The Science of Computer Programming., pp. 231-274, 1987.

[22] W. S. Humphrey, "Managing the Software Process", The SEI Series in Software Engineering, Software Engineering Institute, 1990.

[23] A. Hutcheon, B. Jepson, and J. McDermid, “A Study of High Integrity Ada, Tool Support for High Integrity Software Development”, Ada in Transisition. 1992 Ada UK Internatinal Conference Proceedings, p. 8-30, 1992.

[24] “IEEE Standard for Developing Software Life Cycle Processes,” IEEE Computer Society, April 20, 1992.

[25] J. C. Knight and N. G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Muliversion Programming” IEEE Transactions on Software Engineering, pp. 96-109, Jan. 1986.

[26] J. C. Knight and D. M. Kienzle, “Safety-Critical Computer Applications: The Role of Software Engineering”, University of Virginia, TR-92-93, 1993.

[27] H. K. Knudsen, Reliable Specification and Design Using Linked State Machines, Unpublished, Sandia National Laboratories, June 1994.

[28] Levson, N. G. and Stolzy, J. L., "Safety Analysis Using Petri Nets" IEEE tran. on Software Engineering, SE-13, January 1987, pp. 386-397.

[29] “Software Safety in Embedded Computer Systems” Communications of the ACM, pp. 34-45, 1991.

[30] N. G. Leveson, “An Introduction to Software System Safety,” Tutorial notes from 15th International Conference on Software Engineering, May 1993.

[31] N. G. Leveson, “Requirements Specification for Process-Control Systems,” unpublished, 1994.

[32] N. G. Leveson and C. S. Turner, “An Investigation of the Therac-25 Accidents,” UCI Technical Report #92-108, Info. and CS Dept., University of California at Irvine, November 1992 (also in IEEE Computer, July 1993).

[33] R. C. Linger, H. D. Mills, and B. I. Witt, “Structured Programming: Theory and Practice,” Addison-Wesley, 1979.

[34] B. Littlewood and A. Sofer, "A Bayesian Modification to the Jelinski-Moranda Software Reliability Model", Software Eng. Journal, Vol. 2, pp. 30-41, 1987.

[35] Y. K. Malaiya and P. K. Srimani, Software Reliability Models: Theoretical Developments, Evaluation and Applications, IEEE Computer Society Press, 1990.

[36] H. D. Mills, “Mathematical Foundations for Structured Programming,” IBM TR FSC72-6012, 1972.

[37] J. D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model For Software Reliability Measurement", Proc. 7th International Conference on Software Engineering, pp. 230-237, March 1983.

[38] J. D. Musa, A. Iannino, K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, Inc., 1987.

[39] Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria, National Computer Security Center, Fort George G. Meade, MD, July 1987.

[40] P. G. Neumann, "Illustrative Risks to the Public in the use of Computer Systems and Related Technology," SRI International, Menlo Park, CA, 1993.

[41] P. R. Place and K. C. Kang, "Safety-Critical Software: Status Report and Annotated Bibliography", Carnegie-Mellon Universtity, Pittsburgh, PA, Software Engineering Inst., June 1993.

[42] J. Rushby, "Critical System Properties: Survey and Taxonomy," Reliability Engineering and System Safety, pp. 189-219, 1994.

[43] Sandia Software Guidelines Volume 3, Standards, Practices, and Conventions, SAND85-2346, Sandia National Laboratories, 1986.

[44] K. G. Wika and J. C. Knight, "A Safety Critical Architecture", University of Virginia, CS-94-04, 1994.

[45] G. D. Wyss, "Fault Tree Analysis Methods Overview" presented to Mast Nuclear Safety Oversight Team, Feb. 1994.

Biographies

Elmer Collins

Elmer Collins is a Senior Member of the Technical Staff in the Reliability Assessment Department at Sandia National Laboratories. He is engaged in security fault analysis of use control systems for nuclear weapons. His main research interest is in software reliability. He received his B.S.E.E degree in 1976 from West Virginia Institute of Technology, M.S.E.E degree in 1980 from West Virginia University, and Ph.D. degree in 1985 from West Virginia University.

Larry J. Dalton

Larry holds a BS in Applied Mathematics and an MS in Electrical Engineering from the University of New Mexico. Larry has spent the past 17 years at Sandia National Laboratories in Albuquerque, New Mexico engaged in high-consequence systems development. He has developed personnel entry control systems for DOE high value facilities, an Aircraft Monitor and Control System for gravity nuclear weapons on board the B-52, high performance airborne multiprocessing systems for advanced concepts demonstration and gravity nuclear weapon programmers. He currently manages the Command and Control Software Department at Sandia that develops software and systems safety solutions for high-consequence operations. These activities span nuclear weapons, biomedical applications, transportation, and information security.

David E. Peercy

Dr. Peercy obtained his Ph.D. in Mathematics from New Mexico State University in 1971. He has been involved in many software engineering activities at the national level and has a national reputation in software logistics. He has been an invited speaker and panelist at several national conferences. Dr. Peercy organized and chaired a National Workshop on Software Logistics in 1989; he has been on the program committee for several conferences; and he has been a reviewer for numerous IEEE standards. He has taught short courses/tutorials on computer security, software reliability, software logistics, software quality assurance, and software Independent Verification and Validation. At Sandia National Laboratories, Dr. Peercy has current responsibility for the qualification evaluation of nuclear weapon related software systems and technical leadership for development of a lab-wide Software Management Program. He has also been a primary investigator on a contract with SEMATECH to develop more reliable semiconductor equipment software through use of process improvement methods.

Carl W. Sicking

Carl holds a B.S. degree in Computer Science from Texas A&M University and an M.S.E.E degree in Computer Engineering from the University of New Mexico. Carl has worked at Sandia National Laboratories in Albuquerque, New Mexico for 8 years. His experience in developing high-consequence software includes embedded software for nuclear weapon use control applications.