

Critical Systems and Software Risk to Public Safety: Issues and Research Directions

Shreedevi Inamdar and Hisham M. Haddad
CSIS Department
Kennesaw State University
Kennesaw, GA 30144

Abstract: *Standards concerned with the development of safety-critical systems, and software in such systems in particular, abound today as the software crisis increasingly affects the world of embedded computer-based systems. The risk due to failure of software in safety critical systems leads to severe consequences such as injury or death of humans. Medical instrumentation, railway signaling, and air traffic control are few examples of such systems. Safety-critical systems require the utmost care in their specification, design, implementation, operation, and maintenance. However, few other factors like timeliness, reusability, and cost are impaired. Safety Engineering Process is to identify, eliminate, or control hazards to acceptable levels of risk throughout a system's life cycle. This paper gives a bird's eye view of the impact of software failures on human society, outlines issues and concerns in this area of software engineering, and highlights future research directions to address these issues and concerns.*

Keyword: Safety-Critical Systems, Software Safety, Software Risk Assessment.

1. Introduction

Today, not only do products use computers to operate better or cheaper ("smart" automobiles and appliances are examples), but complex systems are incorporating designs that cannot be operated without computers (for example, unstable aircraft and space vehicles that cannot be operated successfully by humans alone). It is the combination of hardware and software that makes a machine at its best for performance [1]. The software engineering field has secured its place in rapidly growing industries. When such is the case, it is equally important to know the way software can make or break a system's functionality. This work is to investigate, analyze, and highlight the issues and concerns of critical systems involved with the public safety and the risks that software can bring in such systems. We discuss the problems being attacked, attempt to delineate why the problems have not already been solved, and suggest some specific research topics that we believe are of

critical importance in stretching the current limits of complex system.

2. Safety Critical Systems

Safety-critical software systems, found in medical instrumentation, railway signaling, air traffic control, and other type of instruments, are electronic systems in which failure may have severe consequences such as injury or death. These systems require the utmost care in their specification, design, implementation, operation and maintenance, as they could lead to injuries or loss of lives and in-turn results in financial loss [2,3,4]. Decisions about safety-critical systems in a professional and responsible manner are to be made. David Hughes [5] wrote in a recent editorial in Aviation Week and Space Technology: "Information technology is becoming a key part of everything the aerospace and defense industry does for a living, and as the century closes it is computers and software that hold the keys to the future. The [aerospace] industry is being transformed from dependence on traditional manufacturing into something that looks more like IBM and Microsoft with wings."

A central computer with a global positioning system authorizes its movements, says Giras [6], who believes that the rail system is at least as sophisticated, if not more sophisticated, than the airline industry with respect to its reliance on computer systems. Therefore, computer failures "can result in life-threatening situations," he adds. "We're so reliant on IT that we don't spend as much time or concerned with the safety or reliability" of our IT systems, says Haimes [6]. We hear more about failures due to computers: Software errors have resulted in loss of life, destruction of property, failure of businesses, and environmental harm. One of the reasons for the problems is that these systems require that standard engineering techniques be extended to deal with new levels of complexity, new types of failure modes, and new types of problems arising in the interactions between components. All factors like complexity, failures, new types of problems, diagnosing methods and techniques change rapidly with the growing technology and electronics.

Computers exacerbate engineering problems by allowing levels of complexity and coupling with more integrated, multi-loop control in systems containing large numbers of dynamically interacting components along with the software. We are attempting to build systems where the interactions between components cannot be thoroughly planned, understood, anticipated, or guarded against failures or unnatural behaviors. The fundamental problem is intellectual unmanageability: Increased complexity and coupling make it difficult for the designers and developers to consider all the potential system states or for operators to handle all normal or abnormal situations and disturbances safely and effectively. The failures in these systems are arising in the interactions between components and software. The Software and System Safety Research Group is a response to these problems. Its goal is to act as a focus for interdisciplinary research, education, and development to support the engineering and use of computers embedded and controlling complex engineered systems. Software System Safety optimizes system safety in the design, development, use, and maintenance of software systems and their integration with safety critical hardware systems in an operational environment.

3. The Problem

Most control software is too complex for complete mathematical analysis and yet too structured for statistical analysis. At first, heroic human effort, brute force techniques, and tremendous amounts of money were able to get large software projects finished successfully such as the Space Shuttle control system (One of the largest and software development projects of the 1970's, contains about 400,000 lines of code). However, our ambitions are starting to outstretch the limits of what brute force and money can accomplish, and the technology to build such systems and to provide the needed confidence in their quality does not exist. Other complex projects include the U.S. Air Traffic Control System, Space Station Freedom, and commercial and military aircraft contain millions of lines of code. Building such systems requires thousands of people, and just organizing these projects is a massive undertaking.

The result of not solving these system and software engineering problems may be failures in our attempts to build the complex systems of the future. As just one example, the huge cost overruns and technical difficulties encountered in building a new U.S. Air Traffic Control system led to canceling large parts of it a few years ago. The more recent scaled back attempts to provide limited upgrades are also running into problems. We have seen of five satellite launch attempts, several of them blamed on software, including the most recent failure of a Titan IV-B/Centaur Milstar mission.

Producing enormous amounts of code is not enough. The potential for human, environmental, and financial losses with these computer-controlled systems makes quality of paramount importance. Virtually all software will have errors, and we do not currently have the capability to locate and correct all errors. We are putting reliance on products that we cannot demonstrate are trustworthy.

3.1 Why we face these Problems?

Although major initiatives are currently missing, certainly a great deal of effort has been and still is being applied to these problems. Why are we still having trouble building reusable embedded systems? Today we have relatively few problems building the typical software systems. Man's reach always outdistances his grasp as we learn how to build one type of software system successfully; we immediately want to accomplish more. But we cannot blame all our limitations on increasing expectations. Although a large number of researchers have been working on software engineering, their results have had limited use in real systems, there may be several reasons for this.

First, academic researchers have concentrated on the mathematical aspects of problems and solutions while ignoring human factors and the necessarily informal aspects of software development. While mathematical techniques are useful in some parts of the process, informal techniques will always be a large part of any software development effort, and indeed, most engineering projects. Researchers often focus exclusively on formal or informal aspects of software development without considering their interaction. Formalism is crucial in developing software for critical systems, but the limits of modeling reality must be taken into account: (a) the actual system has properties beyond the model, and (b) mathematical methods cannot handle all aspects of system development. No comprehensive approach in developing critical systems will, in the foreseeable future, be entirely formal while informal approaches alone cannot provide adequate confidence. Our approaches must be driven by the need to systematically and realistically balance and integrate mathematical and nonmathematical aspects of software development.

Second, the results of research often lead methodologies that cannot be incorporated into practice by developers and maintainers. Understanding about how to build critical software is not enough. The methodologies must include training and technology transfer and must be usable by those with typical software engineering backgrounds. The methodologies must also incorporate models that are closely related to the problem domain and the way that application experts think about their problems, not necessarily the way that researchers look at the problems. One serious drawback of past and current software engineering research is lack of

scalability. Researchers have developed techniques that work only on small systems. Mathematical techniques have, for the most part, been used only on very limited properties and on unrealistically small problems. There is reason to believe that software development in the large is so different than the toy problems found in most research papers that published techniques that may not apply to real projects. There is urgent need to find a balance of formal and informal techniques that scale by considering, from the start, problems of realistic size and complexity. Software engineering researchers rarely validate their techniques and theories on realistic software. Given the complexity of the systems we are attempting to build, the *only* convincing argument that an approach will work in practice is to validate techniques on real systems.

Third, successfully building of complex systems demands that qualities such as reliability, safety, security, and timing be rigorously addressed and systematically built into the software from the beginning [7]. In addition, simply concentrating on initial development is not enough: These qualities must be preserved as the software evolves during its lifetime. Independent efforts to ensure individual qualities in narrow domains, e.g., security, have made significant progress. However, no approach exists that combines diverse techniques into an *integrated* methodology for developing and maintaining software for critical systems. Furthermore, the methodologies that are developed must be usable by other than their developers and must be incorporated into practices.

4. Specific Areas For Research

Literature search and analysis of gathered data lead to believe that the following areas are of special importance and difficulty in engineering complex systems and thus are appropriate avenues of research. Many of these research goals are at the interface of what has typically been considered software engineering concerns and those of system engineering.

4.1 Modeling and Analysis

Engineers were able to reuse standard designs that had been perfected over many years; most of the new systems using computer control require new designs. The complexity of these systems, does not allow us to build physical prototypes and experiment to improve designs. Instead, mathematical models must be used to verify certain required properties. An important research topic involves powerful and efficient modeling language, analysis techniques to allow prediction and accumulation of information to aid the system, software design, and verification process. Many modeling techniques have

been proposed; most consider very limited system aspects.

Analysis is an intrinsic part of any engineering discipline. Most models are related to single qualities, such as security or reliability. A few general models exist with extensive theories, but these models often lack the power to provide the required information to designers or address the variety of qualities required in complex systems. Most models also provide little help in comparing alternative system designs. We need better formal methods, and ways to interface them to human abilities and to informal methods. The techniques and tools we develop must also be usable by software developers and must be integratable into normal software development environments.

4.2 Engineering for Quality

One of the most important issues in complex systems is achieving and assuring quality. That is, identifying and resolving tradeoffs of various qualities, achieving multiple qualities, providing confidence or assurance of required qualities over the lifetime. The term “quality” is relative. Quality of software depends on the requirements of the system, hardware, and the idea of quality each individual carries. Hence, we have no way to achieve or assure high software quality. Properties (reliability, safety, security, and modifiability) must be built from beginning; they cannot be added or simply measured on existing systems. Up-front planning and changes to the development process are needed to achieve particular objectives. These changes include using notations, techniques, constructing system to achieve particular properties, and validating (at each step) the evolving system has desired properties. Center to this problem is considering the interactions of critical system properties and potential conflicts. An unwarranted assumption is often made that independent approaches to achieving specific software and system qualities can be easily composed, which is false. For example, approaches to ensure usability or reliability may interact in important but indirect ways with approaches to ensure safety properties. Many techniques can be found to attack sub-problems, but these techniques may not be easily integrated or may be costly for each critical property if software development process is not built on the results obtained from previous steps. We need integrated methodologies for developing and maintaining software that encompass the entire development process and consider multiple conflicting goals. A question will arise. “How do we improve the safety factor in the existing system?” The optimal solution will be to get all information on field history such as: hours of use, type of actual use, environmental conditions, experienced personnel, and existing hardware configuration (sensors, computer, I/O).

4.3 Providing Assurance

Significant development effort goes into verification and validation [8]. We are able to execute and test only a fraction of the possible system states before software is put to operation. Yet, particularly for critical systems, high confidence is often a prerequisite for certification or use. Dynamic analysis, i.e., testing, will always have an important place in providing confidence; cost, and criticality are increasing the need for static analysis of software that can provide assurance over the lifetime of software. Testing and analysis should support each other, with testing providing confidence in the correctness of the assumptions made in static analysis. We need to provide more affordable and effective testing, at the same time exploring potentials of static analysis for important properties and understanding the interaction of these two approaches.

4.4 Human-Computer Interaction

Most complex systems require a combination of human and computer control, where humans provide intelligence and problem-solving ability while computers handle aspects requiring speed and computational power. Challenges exist in determining how to allocate tasks between humans or computers and to design the features of this interaction so that the unique capabilities of each are optimized. Simply replacing the human by computers may not result in the most efficient, useful, and safe systems. The desired end is a partnership between the computers and human that is superior to either of them working alone. Serious accidents occur in aircraft and other shared control systems where the design of interaction between computers and humans is being blamed rather than failures or errors of the system components. Although research exists on how to make usable and friendly computer interfaces, very little research exists on how to integrate computers and humans in a complex system. In a slightly different context, a better understanding also is needed to design software tools and languages to minimize the number of errors that are introduced during software development and to provide better tools to software developers. One roadblock in making progress is lack of scientifically established information to make decisions about design of software tools and techniques. There has been a great deal of study of the mathematical and engineering foundations of software engineering, but much less of the psychological foundations.

4.5 Software Evolution

Software engineering approaches often concentrate on initial software development and not on continual evolution of software and its environment. Software is continually changing and evolving, not only because of

discovery of latent errors, but primarily because of changes in the operating environment, for end users and underlying technology. The software must be designed to be changeable without compromising the properties initially verified. Sometimes decisions will have to be made not to change critical software if the risk is high. We need ways to make those decisions. Ways to design and construct software so that software can evolve over time without compromising critical properties and techniques in the evolution process itself.

5. Risk Assessment

As analog and mechanical control systems with measurable risk are being replaced by computers, we need to develop procedures that provide the same level of assurance of acceptable risk, such as, risk of critical infrastructure failures, risk of sudden and severe economic loss, risk of loss of life or disability, risk of loss of public confidence, and risk of loss of our technological edge and leadership.

Software risk assurance and assessment is a key solution to most of these problems. Numerical risk assessments of physical systems usually are derived from (1) historical information about the reliability of individual components and models that define the connections between these components or (2) historical accident data about similar systems. Neither of these assessment approaches applies to software: Historical information is not available, as software is usually customized for every process. Following steps could be helpful:

- Identify assets and identify which are most critical
- Identify, characterize, and assess threats
- Assess the vulnerability of critical assets to specific threats
- Determine the risk (i.e. expected consequences of specific types on specific assets)
- Using Assessments to Identify and Prioritize Risk Reduction Activities
- Identify and characterize ways to reduce those risks

Devising probabilistic models of software reliability is important and very useful in software development. But their usefulness in certifying safety is less clear. The very low failure probabilities and high confidence in these assessments that is required in safety-critical systems require more experience with software than could possibly be obtained in any realistic development process. More important, these models are measuring the wrong thing. Software reliability is defined as compliance with the requirements specification, but accidents most often occur as a result of flawed specifications, i.e., faulty assumptions about the behavior of the environment or the required behavior of the software. Software reliability prediction models assume

that it is possible to predict accurately the usage environment of the software, anticipate and specify the appropriate behavior of the software under all possible circumstances correctly. Both of these goals are impossible to achieve. Just as probabilistic evaluation may not be the most appropriate way to provide confidence in proof of a theorem; it also is not best way to achieve confidence that software will do correct or safest thing under circumstances.

An emphasis on formal and informal verification, analysis, and review may be more appropriate in evaluating a software and system design. We need more research on procedures to identify software-related hazards, to eliminate and control these hazards through design. Apply these safety-analysis techniques during software development to provide confidence in safety of software and aid the design of hazard protection. To evaluate the effectiveness of analysis and design procedures to assess the level of confidence they merit. Qualitative risk assessment and assurance techniques need to be developed if government and society are going to continue to allow the use of computers to control processes that potentially affect public safety.

6. The Insight

Using Assessments to Identify and Prioritize Risk is the best currently know solution. Identify how to reduce risk. Prioritize and decide in what to invest while the software is being developed. Decision makers would have to come to a consensus on which risk reduction strategy to use to set priorities. Starting from the risk assessment:

- High Risk Significant Analyses and Testing Resources
- Medium Risk Requirements and Design Analysis and Dept Test Required
- Moderate Risk High Levels of Analysis and Testing Acceptable with Managing Activity approval
- Low Risk Acceptable

Software can be labeled defective if it does not perform as expected. In turn, defective software can be labeled hazardous if it consists of safety-critical functions that command, control and monitor sensitive systems. Major types being software not executing; software executing too late, too early, suddenly or out of sequence; and software executing but producing wrong information.

The Systematic Software Safety Process consists primarily of the following elements:

- Software safety planning
- Application of the software safety process during all life cycle phases

- Identification and application of life cycle phase-independent software safety activities
- Identification of special provisions
- Software safety documentation

The process (as described in Mil Std 882C) includes a System Safety Program Plan. The plan provides a description of the strategy by which recognized and accepted safety standards and requirements, including organizational responsibilities, resources, methods of accomplishment, milestones, and levels of effort, are to be tailored and integrated with other system engineering functions. Software system safety planning is deemed essential early in the software life cycle. Most importantly, planning should impose provisions for accommodating safety well before each of the software design, coding, testing, and deployment and maintenance phase's starts in the cycle. Moreover, these provisions are to be planned carefully to impact minimally the software development process. The software system safety plan should contain provisions assuring that: Software safety organization is properly chartered and a safety team is commissioned in time.

Acceptable levels of software risk are defined consistently with risks defined for the entire system. A software quality assurance program should be established for systems having safety-critical functions. The software should be analyzed throughout the design, development, and maintenance processes by a software system safety team to verify and validate the safety design requirements have been correctly and completely implemented.

Interfaces between software and the rest of the system's functions are clearly delineated and understood. Software application concepts are examined to identify safety-critical software functions for hazards. Requirements and specifications are examined for safety hazards; while design and implementation are properly incorporated into the software safety requirements. Appropriate verification and validation requirements are established to assure proper implementation of safety requirements. Test plans and procedures can achieve the intent of the software safety verification requirements. The Process lays out a disciplined, systematic methodology that ensures all risks (events and system failures that contribute to expected casualty) are identified and eliminated, or that their probability of occurrence is reduced to acceptable levels of risk. Qualification tests, as referenced in the safety demonstration process and the System Safety Program Plan, are normally conducted to environments higher than expected.

7. Discussion

Those who develop, maintain, and market safety-critical computing systems and components have certain obligations to those who use their systems. The responsibility of engineers for safety derives from clients' and public's right not to be endangered without prior warning in a manner understandable for them. Many factors influence engineers' judgment when deciding which risks are acceptable. Most basic are their value principles regarding what they care about and to what extent. Assessing safety is a complex matter for engineers. For most safety critical systems, when they are developed, those systems must be shown to be acceptably safe. Risk connected to a project or a product needs to be identified, costs of reducing risks is estimated, costs must be weighed against both organizational goals (profit, reputation for quality) and acceptability of risks to clients and public, also project or product need to be tested. Uncertainties in assessing risks arise in all these. For instance, a product is usually tested under controlled conditions. That will not mirror what will happen following mass production and installation under normal operating conditions. The results, finally, are subject to human judgment as to how safe is safe enough, and how much work should be put into making the system safer.

Ethics concerns are essential for safety-critical systems. Ethics is important for managers, designers, and programmers in order to be able to make decisions about safety-critical systems in a professional and responsible manner. When valuation is made during the design process, of product quality attributes, different properties of a product are considered against each other. Hence, ethics is important to decide what quality attributes are to be focused on. In general, ethics constitute a basis of a safety culture within an organization dealing with safety critical systems. An important component of getting ethics foundations is education in professional ethics as a part of computing curriculum. Another important element of assuring appropriate ethical praxis is codes of ethics, which are advisory documents, issued by professional organizations. This analysis addresses also the Precautionary principle, a basic ethical principle that advocates precaution when dealing with systems connected with risks for humans or environment, which is the case with safety critical systems.

Understanding why accidents occurred and making the changes necessary to prevent future accidents requires more than simply identifying proximate cause, human error in transcribing long strings of digits. This type of error is well known and there should have been controls established throughout the process to detect and fix it, including identifying the roll rate filter data as critical through a hazard analysis process. Either these controls

were missing in the development and operations processes or they were inadequately designed and executed. Even though this accident report was unusual in its depth of analysis of the causal factors, the STAMP analysis identified additional questions that might have been asked to provide additional insight into how such accidents might be avoided, particularly at higher levels of development and operations control structures [9]. Identification of coordination problems was a common and important omission; this information provides guidance in creating recommendations for preventing future accidents. In the case of accident, better developments verification, system safety engineering, management, and oversight processes are needed. Design of a comprehensive system safety program that identifies hazards and critical data and then institute's special controls for, oversight and monitoring of software development process seem appropriate. In addition to investigating and analyzing accidents, a systems-theoretic accident model can be used to prevent accidents and to accumulate the information necessary to design for safety during system design and development. Hazard analysis is essentially the investigation of an accident before it occurs. A proactive accident investigation, i.e., hazard analysis, using STAMP rather than the traditional analysis techniques based on event-chain models (fault analysis, event analysis, failure modes and effects criticality analysis) can provide the information necessary to design an integrated socio-technical system, including development and operations, to prevent accidents in software-intensive systems.

8. Conclusion

Accident models provide the basis for safety engineering, both in the analysis of accidents have occurred and in the development of techniques to prevent accidents. To deal with safety in software-intensive and safety critical systems, we will require more sophisticated models than those currently in use. A proposal was made that models based on systems theory would be appropriate. A major difference between a systems-theoretic accident model and a chain-of-events model is that the former does not identify a root cause of an accident. Instead, the entire safety control structure is examined and the accident process to determine what role each part of the process played in the loss. While perhaps less satisfying in terms of assigning blame, the systems theoretic analysis provides more information in terms of how to prevent future accidents.

Safety is closely coupled to the notion of risk. Safety is a subjective measure that makes safety provision and measurement difficult and contentious tasks. Safety concerns in computer systems are even more confusing due to the safety critical nature of system and also due to the software involved. Such systems consist of many

subcomponents that are tightly coupled and have highly complex interactions with both hardware and software. The binding of application to operating system to architecture is a prime example of a tightly coupled system. When such a system is further embedded within larger contexts, such as command and control systems along with software, the probability of failure quickly approaches unity. Myers [10] estimates that there are approximately 3.3 software errors per thousand lines of code in large software systems. This figure is not surprising given that there are many unique end-to-end paths in a moderate-sized program. What is worse is that not all errors in software are created equal, as small errors do not necessarily have small effects, the errors might create different type of accidents and direct or indirect effects on the environment. Some times errors might not seem critical in the initial stage or might remain un-noticed, they in future might bring great accidents or ill effect.

The picture becomes much bleaker when the software/hardware interactions in computer systems are taken into account. In two studies, nearly 10% of all software errors and 35% of all software failures identified were later found to be hardware-related, such as transient faults corrupting data. In fact, it appears that hardware can fail three times as often as software under some circumstances. The most effective means to avoid accidents during a system's operation is to eliminate or reduce dangers during the design and development, not afterwards when the complexity becomes overwhelming, or the people working with the these system or in its environment or are related to the critical system – software smooth functioning should be effectively educated. Safety cannot be considered as an add-on after the system has been developed. Safety must be designed in a system and dangers must be designed out. Moreover, software and hardware safety are inextricably intertwined and must be considered as a whole with special attention paid to the interfaces. Our goals should be the following:

- Safety consistent with mission requirements is designed into the software in a timely, cost effective manner.
- Hazards associated with the system and its software are identified, evaluated and eliminated or the risk reduced to an acceptable level, throughout the lifecycle.
- Reliance on administrative procedures for hazard control is minimized.
- The number and complexity of safety critical interfaces is minimized.
- The number and complexity of safety critical computer software components is minimized. Sound human engineering principles are applied to the

design of the software-user interface to minimize the probability of human error.

- Failure modes, including hardware, software, human and system are addressed in the design of the software.
- Sound software engineering practices and documentation are used in the development of the software.
- Safety issues are addressed as part of the software testing effort at all levels of testing.
- Software is designed for ease of maintenance and modification or enhancement.

9. References

- [1] Paola Bracchi and Vittorio Cortellessa. “A framework to model and analyze the performability of mobile software systems.” *Proceedings of the 4th international workshop on Software and performance*, Redwood Shores, California 2004.
- [2] Marvin Zelkowitz and Ioana Rus, “Understanding IV & V in a safety critical and complex evolutionary Environment: the NASA space shuttle program.” *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001.
- [3] Douglas C. Schmid. “Adaptive middleware: Middleware for real-time and embedded systems.” *Communications of the ACM*, Volume 45 Issue 6, June 2002.
- [4] John C. Knight. “Safety critical systems: challenges and directions.” *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, 2002.
- [5] Software & system safety research group. Nancy Leveson, Aeronautics and Astronautics Massachusetts Institute of Technology.
- [6] Julekha Dash, Freelancer, Lewes, Delaware Published by the Department of Information Technology and Communication (ITC) at the University of Virginia 2002.
- [7] Steve Easterbrook. “Verification and validation of requirements for mission critical systems.” *Proceedings of the 21st international conference on Software engineering*, Los Angeles, California, 1999.
- [8] Nancy G. Leveson. A New Approach to System Safety Engineering. Manuscript in preparation, draft can be viewed at <http://sunnyday.mit.edu/book2.pdf>.
- [9] Jonathan Bowen and Victoria Stavridou. Safety-Critical Systems, Formal Methods and Standards can be viewed at <http://www.jpbowen.com/pub/sfs-sej.pdf>
- [10] The Software system handbook. http://sesam.tranet.fmv.se/arbetsgrupper/IG_Prgsak/Publikat/SSSHandbook.pdf