DOI:10.1145/3615860

Nancy G. Leveson and John P. Thomas

Inside Risks
Certification of Safety-Critical Systems

Seeking new approaches toward ensuring the safety of software-intensive systems.

Software is ubiquitous in safety-critical systems and the size of the software can be enormous—tens of millions of lines of code is not unusual today. How can the safety of such systems be assured? While government certification is only required for a few systems (such as aircraft, nuclear power plants, and some types of medical devices), other commercial products are potentially subject to lawsuits and other liability issues that force companies to have an effective strategy for dealing with safety. At the extreme, highly publicized accidents can lead to a company going out of business. Some military systems, which are often out of the public eye, can have potentially catastrophic consequences if they go wrong.

Almost all current approaches to certifying critical systems or assuring the safety of such systems were created when our engineered products were electromechanical. Meanwhile, enormously complex software has been introduced into these safety-critical systems with relatively few significant changes to the way systems are certified or assured. Software-related accidents in every industry are occurring in exactly the ways assurance approaches determined were implausible or impossible or in ways that were overlooked entirely during certification and safety assessment. Creating effective approaches will require changes in the way software is engineered today.

IMAGE BY MINDTEST

Limitations of Current Certification Approaches

Current standards for certifying safety still largely rely on failure-based and component-based methods (Functional Hazard Analysis (FHA), Fault Tree Analysis (FTA), Failure Modes and Effects Analysis (FMEA), and so forth) developed at least 40–60 years ago, when engineered systems primarily comprised hardware. An assumption was made that accidents were caused by component failures. That assumption was reasonable at the time and led to the idea that high component reliability could assure safety to an acceptable level.

If accidents are assumed to result from failures or human errors, then the standard solution is to design high reliability into products, perhaps by using redundancy, decoupling, monitoring for failures, fault-tolerant or fail-safe designs, and other methods. Humans errors were handled by training and compliance with specified procedures.

The problem with this solution is it does not consider accidents that occur when the components (either physical or functional) work as intended (that is, satisfy their requirements), which is the most common type of accident when software is involved.

Introducing software into the control of safety-critical systems changed the world of engineering, allowing the creation of complex systems not previously possible. At the same time, it created grave difficulties in certifying and assuring the safety of such systems. Before the introduction of software into the control of potentially danger-

ous systems, safety was never an issue for software: It is a pure abstraction and does not explode, catch on fire, or otherwise directly harm humans. It did, however, have the ability to impact critical processes when it started to be used in the monitoring or control of critical systems in the late 1970s, and such use has increased exponentially today. Software undermined the assumptions underlying the current standard system certification and assurance approaches.

Certification of such systems is usually based on the concept of risk. Risk is calculated traditionally by combining the likelihood of a failure combined with the severity of the result. Analysis methods were created to calculate likelihood in engineered systems by combining individual component failures. If the resulting risk figures are not deemed to be acceptable, more redundancy, monitoring, and so forth is built into the design until acceptable risk calculations result. But how does one determine the probability of software behaving in an unsafe way?

There is also the problem of the human operators of these systems. Before software control, human operators were performing relatively simple tasks, such as reading gauges and pushing buttons activating physical controls (such as brakes). Their reliability at performing these simple tasks was assumed to be measurable and predictable. Whether this assumption is true is a matter for debate but irrelevant today as such activities are now being taken over by computers. Humans are performing the much more cognitively complex tasks of supervising and monitoring the computers, for which, like software, there is no way to evaluate the likelihood of an error.

An additional problem arose with the introduction of software in the control of these systems. Underlying the risk calculations are assumptions about the independence of the system components. Software in these systems couples the components in complex and unexpected ways that violate the basic independence assumptions behind the calculations. Also, new, more complex types of human operator errors (such as mode confusion) can be caused by the design of software, which made the problem of risk assessment essentially intractable, although people still try to do it and systems are certified using this approach.

Introducing software into the control of safety critical systems changed the world of engineering.

Some Non-Solutions

As software was added to complex systems, attempts were made to reuse the same certification approaches and solutions that had been used on electromechanical and human system components. However, the same protective design techniques (such as redundancy) are of limited use for software and probabilistic approaches do not apply.

Why are these not a solution to the software system safety problem? A previous *Communications* Inside Risks column discussed what will not work.[2] To briefly summarize: testing and after-the-fact verification; formal methods; reuse of application software;and so on, cannot provide the level of assurance required for safety in complex systems. In addition, software is beginning to be so large and complex itself that after-the-fact assurance is no longer feasible. Applying standard hardware design techniques such as redundancy, fault-tolerance, and others turned out to be much more difficult for software than for hardware.

Attempts have been made to solve the problem by improving the way software is developed. The most common has been to require increased "development rigor" such as ensuring the source code complies with programming standards, the software architecture is verifiable, and verifying that software meets requirements. Concepts such as level of rigor (LOR), design assurance levels (DAL), and safety integrity level (SIL) were created.

The problem is that simply increasing development rigor does not prevent the most important software problems in complex systems today: flawed requirements. Software can and has operated in unsafe ways despite exhibiting 100% reliability with respect to satisfying its specified requirements. In fact, many, if not most, software-related accidents have occurred when the software operated exactly as it was designed and specified to operate, that is, the software successfully implemented the requirements given to the software engineers. In these cases, the requirements were flawed.

A few standards deal with software by requiring generic activities such as "identify errors" with no guidance on how this can be accomplished or what is considered good enough. They end up with essentially useless generic checklists, such as "are the requirements complete?" or "are the requirements traceable?" with no way to look any deeper.

Formal methods will not solve the problem, although it is often posited they will. While formal methods can potentially show the consistency of software with its requirements, there exists no formal model of engineered systems (particularly one using discrete math) and the humans who operate them that can be used in this mathematical exercise. Simply showing the consistency of requirements with code does not solve the most important problems. Tony Hoare recognized this limitation in a presentation at a workshop in 1996 just before his retirement, stating: "Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalisation to solve the problems of reliability that arise when programs get large and more safety-critical. Programs have now gotten very large and very critical—well beyond the scale which can be comfortably tackled by formal methods. There have been many problems and failures, but these have nearly always been attributable to inadequate analysis of requirements or inadequate management control. It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve."[1]

AD TK

Continuing to teach students that formal methods are the solution to every problem is not going to solve the most important problems related to safety. The field of computer science and software engineering needs to expand what are considered to be academically acceptable solutions to complex problems.

Faced with the dilemma of assuring the safety of software control, a few engineers have tried to use the same hazard analysis techniques developed to identify critical hardware failures but to apply them to software. The problem they ran into was that these techniques assume that accidents occur only because of "failures"—including functional failures—and that these failures can be assigned probabilities. Those assumptions do not apply for software (or humans).

The fundamental flaw in the current approach is that adding software control in engineered systems changes the nature and cause of accidents beyond the concept of simple failures. As stated earlier, the software or other components may be operating exactly as intended and specified, but the specified requirements may be flawed and lead to an accident. Therefore, the traditional failure-based hazard analysis techniques do not apply and entirely new ones are needed.[3]

What Are Some Potential Solutions?

Old approaches to creating and assuring safety-critical software do not scale to today's complex systems. Something new is needed. What that might be needs to be the focus of new research, but after working on safety-critical systems for decades, we have some ideas about promising directions to pursue. Progress will require new ideas in many areas.

More emphasis on the up-front development effort. Much more effort is needed in the early stages of complex systems and the software contained in them. While skipping or deferring the planning steps such as concept development, requirements specification, and documentation practices may be appropriate for short-lived, non-critical, and relatively small software, they introduce enormous vulnerabilities when used in large, complex safety-critical systems and can result in more rework in the long run. The result may be not only less safe, but more costly. New, more powerful techniques are needed for the planning stages of development (both at the system and software level) as well as new approaches for managing large, long-lived, critical projects. The cost and schedule overruns common today in such systems is not sustainable over the long term. It simply encourages the skipping or watering down of critical steps to prevent such occurrences.

New approaches to identifying system safety requirements. Writing requirements down in a formal language may support efforts to verify that software meets a set of formal requirements, but it does nothing to address the single largest contributor to software-related accidents—flawed requirements. In fact, formal requirements specification languages may degrade it by making the requirements more difficult to review, validate, and identify underlying assumptions. In addition, communication with interdisciplinary experts, who may recognize critical problems of which software specialists are unaware, is inhibited by using formal specification languages. We need rigorous specification languages that are understandable and reviewable by experts of various types. How can requirements specifications be derived from and easily traced to the hazard analysis methods used to identify hazardous system behavior?

System and software engineering methods to design in safety from the beginning. Safety cannot be argued in or assured after the fact, and we need to stop promoting or relying on such approaches. Complex systems must be designed to be safe from the very beginning of development.

Many certification and engineering processes wait until after design and implementation to perform validation. While early engineering decisions can have the largest impact on safety, they are difficult or impossible to change late in the development process. Performing validation after design and implementation not only drives enormous rework costs, but it also creates strong incentives during validation to find minor patches that can be argued to be "safe enough" instead of the strongest, most effective solutions that may require more rework because they

were discovered late during validation.

We need better ways to identify up front the behavioral safety requirements and constraints for the system as a whole and then to allocate them to the system components. If some of those components are implemented by AI, how will it be assured that the AI software implements its safety requirements? How can system and software architectures be created that will assist in ensuring that the safety requirements are satisfied? Are there architectural design techniques that allow ensuring that the architecture, if implemented correctly, will enforce the safety-related requirements?

Another part of the answer to designing safety into a system lies in developing rigorous specification languages that are understandable and reviewable by all the system experts. Model-Based System Engineering (MBSE) could be part of the solution, but only if multiple modeling languages are developed and used. A model necessarily involves ignoring some aspects of a design. The currently most popular MBSE modeling language cannot be used for sophisticated and powerful hazard analysis because it omits some of the most important characteristics and relationships that affect safety.

New software design approaches. Simplicity is important for software as is sophisticated documentation and engineering support tools. Rube Goldberg inspired designs are not appropriate for systems with hundreds of millions of lines of software. While we have always recognized the importance of coupling and cohesion in software designs, much more powerful design and analysis techniques are needed than just these. Programming in the small is becoming less and less relevant as sophisticated code-generation tools are created. What other types of software design approaches may be more useful at the "programming in the large" level? Is object oriented design appropriate for large-scale control software? Might a new "control-oriented design" approach be better? In general, should all design use the same method, or are different design methods needed for software in different types of systems?

Integrating hman factors into system and software development. The role of human operators is changing from direct control to monitoring the operation of computers that control the system components. At the same time, the complexity of our systems is overwhelming the capability of humans to operate them. While total autonomy is a reasonable goal for many relatively simple systems, taking human ingenuity, problem solving, flexibility, and adaptability out of our complex systems will lead to disaster. The design of application software is further contributing to the problems by confusing operators and leading to fatal mistakes—even when the software operates exactly as designed (no failures). Rather than simply trying to eliminate humans or blaming them for the accidents that result from flawed software, software and system engineers must be trained in sophisticated human factors and work closely with experts in the design of the total system, not just the design of the software components. New modeling and analysis techniques are needed to support this integrated design and analysis. The new techniques will need to consider complex interactions, such as software-induced human error, and the techniques must be applicable during early development before the software is designed and before simulators have been created for testing. The techniques will need to be usable by a diverse, interdisciplinary team of experts, not only by software experts who are fluent in a particular formal requirements language.

Simply increasing development rigor does not prevent the most important software problems in complex systems today: flawed requirements.

New, more powerful hazard analysis methods. Because traditional failure-based hazard analysis methods do not apply to software, new more sophisticated hazard analysis is needed. These new analysis approaches must involve more than just identifying failures.[3] The way we analyze and assure safety needs to reflect the state of engineering today, not that of 50 years ago. We have created a hazard analysis technique (called STPA) based on a new assumption that losses result from unsafe control over system and component behavior rather than simply component failures.[4] The rapid and extensive adoption of STPA by industry and development of standards demonstrates the need for newer and more powerful hazard analysis methods that include hardware, software, operators, management, etc. How can STPA be improved or extended? Can new and improved modeling and analysis techniques based on our theory of control or a different underlying theory be created? Are there other, even better, general approaches to creating more powerful hazard analysis techniques?

New, interdisciplinary system engineering techniques: We need interdisciplinary development methods to create safe systems. We cannot effectively solve these complex problems through decomposition—with human factors experts analyzing one small part (the operational procedures and interfaces), software experts analyzing one small part (the software), and other engineers handling the rest. Working separately on different parts of the system, these groups all use their own methods and models. Even if they wanted to communicate, they can't do it effectively because they are all using different languages and tools. This is not a viable long-term approach to system engineering for the complex systems of today and the future. We need methods that enable multiple disciplines to collaborate effectively together on the whole system.

New certification approaches: The certification approaches devised before software became ubiquitous need to be changed. Small tweaks that essentially treat software as the same as hardware will not work. How should the software-intensive systems of the future be certified? How can and should safety be assured without using the probabilistic approaches of the past that are no longer feasible?

Improved management of change.

AD TK

Many certification and engineering processes wait until after design and implementation to perform validation.

Almost all accidents occur after some type of change. At the same time, systems and their environments change continually during operation. These changes may be planned or unplanned. Planned changes are the most easily handled. Management of change procedures are commonly found in industry. But even if the change is planned (for example, an upgrade or new version of the system), changes in software that contains tens of millions of lines of code raises the problem of how to assure that the change has not introduced potentially dangerous behavior in some indirect way? One part of the solution is the identification (and recording) of design rationale and assumptions about the system and its environment. Different design and development techniques will be needed for sustainability of critical software.

Unplanned changes are even more difficult to handle, for example unanticipated changes in the hardware, human behavior, or the environment. Systems often migrate toward states of higher risk during use. Leading indicators are needed to identify when a change in the system or its environment that could be critical has occurred. In addition, improved methods for making such changes are required. Traceability is part of the answer, where the hazards and design solutions identified early in the system development can be traced to the parts of the code that are affected and vice versa.

Conclusion

After-the-fact assurance and certification is no longer practical or even feasible for the large, complex, critical systems being created today. New approaches to creating, assuring, and certifying software-intensive systems are needed.

Safety is a system engineering problem, not a software engineering one. But the system engineering solutions cannot be created or implemented without the participation of software engineers: that is, software/hardware/human factors engineers working together rather than in silos. Perhaps we should not be training individuals to separately fill these roles.

The solution to the problem is likely to involve changes to standard software engineering approaches and definitely changes to education and training. New models and analysis methods, new architectural and design approaches, and more up-front work before generating software rather than depending on post-construction validation will be required. We need to design safety into systems from the very beginning of development, not depend on post-design assurance. This will require that software engineering become a true subdiscipline of system engineering and not just a glorified name for generating code. Software engineers will need to work hand-in-hand with system engineers and human factors engineers to create acceptably safe and secure systems comprising software, hardware, and humans. This goal will require enormous changes in education and practice, which presents new and important challenges for software engineering research. C

References

1. Hoare, C.A.R. Unification of theories: A challenge for computing science. In *Proceedings of the 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop on Recent Trends in Data Type Specification*, Springer-Verlag, (1996), 49–57.
2. Leveson, N. Are you sure your software will not kill anyone? *Commun. ACM 63*, 2 (Feb. 2020).
3. Leveson, N. *Engineering a Safer World.* MIT Press, 2012.
4. Leveson, N. and Thomas, J. *STPA Handbook*; https://bit.ly/3QRRggH

Nancy G. Leveson (leveson@mit.edu) is Jerome C. Hunsaker Professor in Aeronautics and Astronautics in the Engineering Systems Laboratory at the Massachusetts Institute of Technology, Cambridge, MA, USA.

John P. Thomas (jthomase4@mit.edu) is a co-director of the Engineering Systems Lab and Executive Director of the Safety and Cybersecurity Research group at the Massachusetts Institute of Technology, Cambridge, MA, USA.

Copyright held by authors.