System Safety in Computer-Controlled Automotive Systems

Nancy G. Leveson

Massachusetts Institute of Technology
leveson@mit.edu

1 Introduction

Computers are quickly taking over safety-critical functions in transportation systems. Not surprisingly, we are starting to experience incidents and accidents related to the software components in these systems, including a recent recall related to ABS failure. Software allows unprecedented complexity and coupling in these systems, and these factors are stretching our current engineering techniques for assuring acceptable risk. This paper summarizes the state of the art in software system safety and suggests some approaches possible for the automotive and other industries.

2 Types of Accidents

The reliability engineering and system safety engineering communities differ in their approach to safety. Whereas reliability engineering concentrates on component failure accidents, system safety deals with a broader class of accidents including both component failure and system accidents, a new type of accident becoming important in complex, computer-controlled systems.

2.1 Component-Failure Accidents

The reliability engineering approach to safety rests on the assumption that accidents are caused by component failure(s). The obvious solution, given this assumption, is to increase the reliability of the components and to build fault tolerance into the system design to account for these failures. The approach is very effective when accidents are primarily caused by component failure.

Many industries, in addition to increasing component reliability, apply a "fly-fix-fly" approach where accidents are investigated after they occur and the lessons learned are applied to modifications of existing systems and to the design of future ones. In this way, designs are systematically improved over time. This approach is effective in industries where designs and technology are relatively stable over time, but it is less effective when new types of systems are built or the introduction of new technology leads to radically new designs for old systems.

2.2 System Accidents

The introduction of new technology, especially digital technology and the increasing complexity of the designs (most of it made possible by the use of computers) is starting to produce a change in the nature of accidents. While accidents related to hardware failure are being reduced, *system accidents* are increasing in importance.

System accidents arise in the interactions *among* components (electromechanical, digital, and human) rather than the failure of individual components [Per84]. For example, recently the FAA issued an aircraft AD or advisory directive (similar to a recall in the auto industry) that cited the possibility of a delayed stall warning if the stall conditions occurred while the flaps were moving. In this case (as is true for most system accidents), each component worked according to its specification, but subtle component interactions at the system level allowed the stall warning to be delayed.

System accidents are related to interactive complexity and tight coupling. The underlying factor is intellectual manageability. A "simple" system has a small number of unknowns in its interactions within the system and with its environment. A system becomes complex or intellectually unmanageable when the level of interactions reaches the point where they cannot be thoroughly planned, understood, anticipated, and guarded against. Introducing new technology increases the problems by introducing more unknowns into the design, and, in the case of digital technology, potentially more interactions. System accidents arise when interactive complexity reaches the point where it is difficult for designers to consider all the potential system states or for human operators to handle all normal and abnormal situations and disturbances safely and effectively.

The second factor in system accidents, tight coupling, allows disturbances in one part of a system to spread quickly to other parts. We are using computers to build systems with more integrated, multi-loop control of large numbers of dynamically interacting components. When almost any part of a system can potentially affect any other part, the problems of predicting and controlling those interactions quickly overwhelms our current engineering techniques and mental abilities.

As the interactive complexity and coupling in our system designs has increased, so too have system accidents.

3 The Role of Software in Accidents

The increasing use of software is closely related to the increasing occurrence of system accidents. Software usually controls the interactions among components and allows almost unlimited complexity in component interactions and coupling compared to the physical constraints imposed by the mechanical linkages replaced by computers. The constraints on complexity imposed by nature in physical systems do not exist for software and must be imposed by humans on their design and development process.

Indeed, computers are being introduced into the design of virtually every system primarily to overcome the physical constraints of electromechanical components. The problem is that we have difficulty reining in our enthusiasm for building increasingly complex and coupled systems. Without physical constraints, there is no simple way to determine the point where complexity starts to become unmanageable.

This feature of software might be called the "curse of flexibility": It is as easy—and probably easier—to build complex software designs as it is to build simple, clean designs. And it is difficult to determine the line where adding functionality or design complexity makes it impossible to have high confidence in software correctness. The enormous state spaces of digital systems means that only a small part can be exercised before the system is put into operational use; thus the confidence traditionally obtained through testing is severely limited. In addition, humans are not very good at self-imposed discipline (versus the discipline imposed by nature in physical systems): "And they looked upon the software, and saw that it was good. But they just had to add this one other feature ..." [McC92].

Computers are also introducing new types of failure modes that cannot be handled by traditional approaches to designing for reliability and safety (such as redundancy) and by standard analysis techniques (such as FMEA). These techniques work best for failures caused by random, wear-out phenomena and for accidents arising in the individual system components rather than in their interactions.

We are even witnessing new types of human errors when interacting with or within highly-automated systems. Although many of the accidents in high-tech systems such as aircraft are being blamed on the pilots or controllers, the truth is that these systems are often designed in such a way that they are inducing new types of human behavior and human error: The problems are not simply in the human but in the system design. A recent report on pilot errors in high-tech aircraft describes some of these system design problems [BAS98] as have others, e.g., [Lev95, LRK97, SWB95]. The situation is even more serious and potentially difficult to solve in the auto industry where drivers are not highly trained and carefully selected as are commercial pilots.

Using the traditional approach to safety, we would simply try to increase the reliability of the software and introduce software fault tolerance techniques. Unfortunately, this approach will not work for software because accidents related to software are almost never related to software unreliability or failure of the software to satisfy its specification [Lev95, Lut92]. Rather, these accidents involve software that correctly implements the specified behavior but a misunderstanding exists about what that behavior should be. Software-related accidents are usually related to flawed software requirements—not coding errors or software design problems.

In the past two decades, almost all software-related accidents can be traced to flawed requirements in the form of (1) incomplete or wrong assumptions about the operation of the controlled system or required operation of the computer or (2) unhandled control-system states and environmental conditions. For example, an aircraft weapons management system was designed to keep the load even and the plane flying level by balancing the dispersal of weapons and empty fuel tanks. Even if the plane was flying upside down, the computer would still drop a bomb or a fuel tank, which then dented the wing and rolled off. In an F-16 accident, an aircraft was damaged when the computer raised the landing gear in response to a test pilot's command while the aircraft was on the runway (the software should have had a weight-on-wheels check). One F-18 was lost when the aircraft got into an attitude that the software was not programmed to handle. Another F-18 crashed when a mechanical failure caused the inputs to the computer to arrive faster than was expected, and the software was not designed to handle that load or to fail gracefully if the original load assumption was not satisfied. Accidents in the most highly automated commercial aircraft, the A320, have all been blamed on pilot error, but an even stronger argument can be made that the design of the automation was the primary culprit and induced the human errors.

Merely assuring that the software satisfies its requirements specification or attempting to make it more reliable will not make it safer when the primary cause of software-related accidents is flawed requirements specifications. In particular, software may be highly reliable and correct and still be unsafe when:

- The software correctly implements its requirements but the specified behavior is unsafe from a system perspective;
- The requirements do not specify some particular behavior required for the safety of the system (i.e., the requirements are incomplete); or
- The software has unintended (and unsafe) behavior beyond what is specified in the requirements.

As noted, almost all accidents related to software have involved one of these requirements flaws. Ensuring that the software satisfies its requirements will not prevent these accidents.

In periods of rapid technology infusion (as the automotive industry is now experiencing in the form of new digital technology) and new or rapidly changing design, the fly-fix-fly and reliability engineering approaches need to be augmented with techniques that predict and ameliorate both traditional accident causes related to the use of new technology and new accidents causes *created* by the new technology.

One possible way forward is the use of system safety engineering techniques, which were developed to deal with system accidents in complex defense systems. But even these system engineering techniques will need to be extended to deal with the new levels of complexity, new types of failure modes, and new types of problems arising in the interactions between components in our new system designs.

4 System and Software Safety

System Safety as a discipline stems from safety concerns about the first ICBM (Inter-Continental Ballistic Missile) systems in the

1950's. Although great emphasis in engineering these systems was on safety (as they carry nuclear weapons and are highly explosive and dangerous in themselves), an unacceptable level of accidents and incidents resulted. The lack of pilots meant human operators could not be blamed for the accidents, as had been standard practice for military aircraft accidents. The root cause of the ICBM safety problems stemmed from the unprecedented complexity in these systems and the integration of new, advanced technology. System safety engineering was developed to cope with these new factors and was part of the related emergence of system engineering as an identified engineering discipline at the same time.

System safety involves applying special management, hazard analysis, and design approaches to prevent accidents in complex systems. It is a planned, disciplined, and systematic approach to preventing or reducing accidents throughout the life cycle of a system. Rather than relying only on learning from past accidents or increasing component integrity or reliability, an attempt is made to predict accidents before they occur and to build safety *into* the system and component designs by eliminating or preventing hazardous system states.

The primary concern in system safety is the management of hazards. A *hazard* is a system state that will lead to an accident given certain environmental conditions beyond the control of the system designer. An uncommanded behavior of an automobile steering system, for example, may or may not lead to an accident, depending on the conditions under which it occurs and the skill of the driver. However, if worst case environmental conditions could occur, then eliminating or controlling the hazard will increase safety.

In system safety engineering, as defined in Mil-Std-882, hazards are systematically identified, evaluated, eliminated, and controlled through hazard analysis techniques, special design techniques, and a focused management process. In this approach to safety, hazard analysis and control is a continuous, iterative process throughout system development and use. Starting in the earliest concept development stage, system hazards are identified and prioritized by a process known as Preliminary Hazard Analysis (PHA). Safety-related system requirements and design constraints are derived from these identified hazards.

During system design, system hazard analysis (SHA) is applied to the design alternatives (1) to determine if and how the system can get into hazardous states, (2) to eliminate hazards from the system design, if possible, or to control the hazards through the design if they cannot be eliminated, and (3) to identify and resolve conflicts between design goals and the safety-related design constraints.

The difference between this approach and standard reliability engineering approaches is that consideration of safety at the system design stage goes beyond component failure; the analysis also considers the role in reaching hazardous system states played by components operating *without* failure. SHA considers the system as a whole and identifies how possible interactions among subsystems and components (including humans) as well as the normal and degraded operation of the subsystems and components can contribute to system hazards. For example, not only would the effect of normal and degraded or incorrect operation of the steering subsystem of an automobile be considered, but also potential hazard-producing interactions with other components such as the braking subsystem.

If hazards cannot be eliminated or controlled satisfactorily in the overall system design, then they must be controlled at the subsystem or component level. In subsystem hazard analysis (SSHA). the allocated subsystem functionality is examined to determine how normal performance, operational degradation, functional failure, unintended function, and inadvertent function (proper function but at the wrong time or in the wrong order) could contribute to system hazards. Subsystems and components are then designed to eliminate or control the identified hazardous behavior.

The results of system and subsystem hazard analysis are also used in verifying the safety of the constructed system and in evaluating proposed changes and operational feedback throughout the life of the system.

While this system safety approach has proven to be extremely effective in reducing accidents in complex systems, changes are required for the new software-intensive systems we are now building.

5 The Safeware Methodology

The Safeware methodology extends the basic system safety engineering process to handle digital components and subsystems [Lev95]. In this approach to building safety-critical systems, instead of simply trying to get the software correct and assuming that will ensure system safety, attention is focused on eliminating or controlling the specific software behaviors that could lead to accidents. Potentially hazardous software behavior is identified in the system and subsystem hazard analyses. The information derived from these analyses is used to ensure that (1) the software requirements are complete and specify only safe behavior and (2) the entire software development and maintenance process eliminates or reduces the possibility of the unsafe behavior.

The software safety activities in this methodology are all integrated into and a subset of the overall system safety activities. Emphasis is on building required system safety properties into the design from the beginning rather than relying on assessment later in the development process when effective response is limited and costly.

Building safety into software requires changes to the entire software life cycle:

- **Project Management:** Special project management structures must be established, including assigning responsibility for software safety and incorporating it into the software development process. The Software Safety Engineer(s) will interact with both the software developers and maintainers and with the system engineers responsible for safety at the system level.

- **Software Hazard Analysis:** Software hazard analysis is a form of subsystem hazard analysis used to identify safety-critical software behavior, i.e., how the software could contribute to system hazards. The information derived from this process, along with the system safety design constraints and information from the system hazard analysis, is used to: (a) develop software safety design constraints, (b) identify specific software safety requirements, (c) devise software and system safety test plans and testing requirements, (d) trace safety-related requirements to code, (e) design and analyze the human–computer interface, (f) evaluate whether potential changes to the software could affect safety, and (g) develop safety-related information for operations, maintenance, and training manuals.

- **Software Requirements Specification and Analysis:** Because software requirements flaws, and in particular, incompleteness, are so important with respect to software safety, it is critical that blackbox software requirements (required functionality, including performance) be validated with respect to enforcing system safety design constraints and to satisfying completeness criteria.

- **Software Design and Analysis:** The software design must reflect system safety design contraints (1) by eliminating or controlling software behavior that has been identified as potentially hazardous and (2) by enforcing system safety design constraints on the behavior and interactions among the components being controlled by the software. Note that even if the specified software behavior is safe, simply implementing the requirements correctly is not enough—the software can do more than is specified in the requirements, i.e., there is a potential problem with unintended software function or behavior in addition to specified behavior.

- **Design and Analysis of Human–Machine Interaction:** System hazards and safety-related design constraints must be reflected in the design of human–machine interaction and interfaces. The software behavior must not induce human error and should reduce it when possible.

- **Software Verification:** The software design and implementation must be verified to meet the software safety design constraints and safety-related functional and performance requirements.

- **Feedback from Operational Experience:** Feedback sources must be established and operational experience used to ensure both that (1) the analysis and design were effective in eliminating software-related hazards and (2) changes in the environment or use of the system have not degraded safety over time. The assumptions of the system and software hazard analysis can be used as preconditions on and metrics for the operational use of the system.

- **Change Control and Analysis:** All changes to the software must be evaluated for their potential effect on safety. Usually, someone responsible for safety will sit on the software configuration control board. Changes must also be reflected in updates to all safety-related documentation.

The actual implementation of the Safeware methodology within a particular company or project should be optimized for that particular environment and product characteristics. One process model will not fit all situations. Some sample tasks that might be included in such a tailored process are:

- Trace identified system hazards to the software interface.

- Translate identified software-related hazard into requirements and constraints on software behavior.

- Analyze software requirements for completeness with respect to safety-related criteria.

- Develop a software hazard tracking system.

- Incorporate safety into the configuration control system including the decision-making process.

- Design the software to eliminate or control hazards. Trace safety-related requirements and constraints to the code.

- Review all test results for safety issues and trace identified hazards back to the system level.

- Plan and perform software safety testing and formal or informal walkthroughs or analysis for compliance with safety requirements and constraints.

- Perform special safety analysis on human–machine interaction and interfaces and on any special safety-related architectural features such as separation of critical and non-critical software functions.

- Include safety information and design rationale in software design documentation, user manuals, maintenance manuals, etc. and update it as changes are made.

Although it might appear that these added tasks will substantially increase the cost of producing software, increased overall software and system development costs do not necessarily follow from use of the Safeware methodology. At least half the resources in software development today are devoted to testing. For safety-critical systems, much of this testing is used to build confidence that the software is safe. Unfortunately, the confidence that can be achieved this way is limited—we are able to test only a small fraction of the enormous state space of most digital systems (or combined digital–analog systems)[1].

The Safeware approach provides a way out of this dilemma by potentially reducing (but not eliminating) the testing needed to provide the same or even increased levels of confidence. In addition, automated tools can be used to assist

[1] As an example, our model of the collision avoidance logic in TCAS II, an airborne collision avoidance system required on virtually all commercial aircraft in the U.S., has approximately 10^{40} states.

the system engineer and software engineer in performing the additional tasks.

6 The SpecTRM Toolset

The Safeware methodology can be integrated into any system and software development environment. But the complexity of the systems we are building often necessitates the use of special tools to assist designers and the integration of these tools into the system and software development environment.

For the past 20 years, my students and I have been experimenting with techniques and tools to implement the Safeware approach. Our safety and requirements analysis techniques, along with new specification languages and tools, are part of an experimental design and development environment called SpecTRM (Specification Tools and Requirements Methodology) [LR98]. The goal of SpecTRM is to support the design, implementation, and maintenance (evolution) of complex, safety-critical systems. SpecTRM is a general system and software engineering development environment with safety and hazard analysis processes and information tightly integrated into the process and engineering environment in which safety-related decisions are made.

The system specifications in SpecTRM, called *intent specifications*, use a new way of structuring specifications based on fundamental research in human problem solving [Lev99]. Intent specifications include complete traceability from high-level requirements and hazard analyses to code. The support for traceability from hazard analysis to design and implementation is valuable in both designing safety into the system and ensuring that safety is not compromised during operational use and system evolution.

Intent specifications can be analyzed for safety, both formally and informally. This analysis does not require separate modeling languages or models—the appropriate parts of intent specifications (i.e., the required blackbox behavior of the components, including components implemented in software) are executable and usable for both formal analysis and simulation (including hardware-in-the-loop simulation).

Because the specifications are executable and can serve as rapid prototypes, most of the problems that plague traditional code-based prototyping disappear. The specification is always consistent with the functionality of the prototype because it *is* the prototype. Once the design has been completed, the validated executable specification can be used for generating the real system. In addition, because the executable specification is based on an underlying formal state-machine model, at any time the dynamic evaluation (execution or simulation of the specified behavior) can be augmented with formal analysis. The language used to specify blackbox requirements in intent specifications, SpecTRM-RL (SpecTRM Requirements Language), was designed to be easily used by engineers without extensive training in programming or obscure mathematical notations [LHR99].

In addition to the specification languages and tools, various types of analysis tools are or will be included in SpecTRM to assist the system designer including visualization and simulation tools, hazard analysis tools, robustness and fault tolerance analysis tools, requirements analysis tools (including support for completeness analysis), human-error analysis tools (to detect design features that may contribute to mode confusion and other errors related to situation awareness), automated generated of test data from the requirements specification, code generation from SpecTRM-RL specifications, and training tools for system operators.

Such support will be necessary if we are to stretch the limits of intellectual manageability and complexity in the safety-critical systems we build.

7 Conclusions

The automotive industry is quickly increasing the complexity and coupling in their new vehicle designs. This complexity is starting to outstrip our ability to provide high confidence that unexpected interactions between components will not lead to accidents. Software and digital systems provide great challenges as exhaustive testing and standard approaches to high component reliability are infeasible and ineffective in reducing risk.

Other industries have found that almost all software-related accidents are related to errors in the requirements, not coding errors or software "failure." The problems really relate to *system* engineering and the role that computers are playing within the system as a whole and, therefore, system engineering solutions will be required. To build increasingly complex systems with acceptable risk using software components, we will need to introduce new approaches to both system and software engineering and to project management. This paper has suggested some forms these changes might take.

References

[BAS98] Bureau of Air Safety Investigation. Advanced Technology Aircraaft Safety Survey Report. Dept. of Transport and Regional Development, Australia, 1998.

[Lev95] Leveson, N.G. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, 1995.

[Lev99] Leveson, N.G.. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. on Software Engineering*, accepted for publication (to appear in 2000).

[LHR99] Leveson, N.G., Heimdahl, M.P.E., and Reese, J.D. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. *ACM/Sigsoft Foundations of Software Engineering/European Software Engineering Conference*, Toulouse, September 1999.

[LR98] Leveson, N.G. and Reese, J.D. SpecTRM: A System Engineering Environment. *Digital Avionics Systems Conference*, Seattle, 1998.

[LRK97] Leveson, N.G., Reese, J.D., Koga, S., Pinnel, L.D., and Sandys, S.D. Analyzing requirements specifications for mode confusion errors. *Workshop on Human Error, Safety, and System Development*, Glasgow, 1997.

[Lut92] Lutz, R.R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 35–46, January 1993.

[McC92] McCormick, G.F. When reach exceeds grasp. Unpublished essay.

[Per84] Perrow, C. *Normal Accidents: Living with High-Risk Technology.* Basic Books, Inc., New York, 1984.

[SWB95] Sarter, N.D., Woods, D.D., and Billings, C.E. Automation surprises. In G. Salvendy (ed.), *Handbook of Human Factors/Ergonomics*, 2nd Edition, Wiley, New York, 1995.