Using COTS Components in Safety-Critical Systems*

Nancy G. Leveson
Software Engineering Research Lab
Aeronautics and Astronautics Dept.,
Massachusetts Institute of Technology
Cambridge, MA 02139
U.S.A.
leveson@mit.edu

1 Introduction

Risk is a broadranging and multidimensional topic, including both management risks and technical risks. Management risks for COTS are well known, such as loss of market control, rapid obsolescence, and the shift from a buyer's market to a seller's market. Technical risk factors are less well understood. These factors include interoperability and performance issues as well as safety. This paper concentrates on risks related to safety, where safety is defined broadly as related to a significant loss (accident) involving human life or health, environmental damage, money, or system mission. The risk becomes a safety issue when the loss is significant enough that it becomes necessary or worthwhile to devote resources to reducing the risk.

One of the major drivers for using COTS software is to save money. Much, if not all, of the savings, however, may be offset by the activities needed to ensure an acceptable level of risk. This assurance might involve additional testing or analysis procedures. In some highly critical systems, COTS may raise the cost of certification or ensuring safety to the point where the use of COTS products is no longer feasible or cost-effective; any potential savings are eliminated by additional assurance costs.

Before any conclusions can be reached about the cost of achieving acceptable safety risk using products or components involving COTS software, we need to define what is meant by safety and the type of accidents being considered, the process required to achieve acceptable risk for those accident types, and the potential effect of using COTS on that process.

*This work was partially supported by a grant from the NASA IV&V Facility in Fairmont, West Virginia. The paper was a keynote address at the RTO meeting on Commercial Off-The-Shelf Products in Defense Applications held in Brussels, April 2000.

2 System Safety Engineering Overview

The approach one takes to assuring acceptable safety is driven by the underlying model of accident causation used: How one views the etiology of accidents in turns drives any prevention strategy. The increasing use of software to control potentially dangerous systems is complicating things further by changing the basic nature of the accidents that are occurring in these systems.

2.1 Types of Accidents

Because considering the types or general causes of accidents is critical in determining the appropriate approach to preventing them, any answer to the COTS risk question must first start from a basic understanding of accident causation and prevention. Engineers have long dealt with component failure accidents where the accident arises from component failure, i.e., behavior that does not satisfy specified or expected behavior. The events associated with the accident may involve multiple failures or cascading failures, but in all cases the initiating cause is the failure of a component to achieve its goals.

The most common approach to preventing such accidents involves one or both of the following:

1. Reducing or preventing the component failure by increasing the reliability or life of the component or reducing the stress under which the component operates,
2. Protecting the overall system operation from being adversely affected by the failure of the component(s).

The second approach is often achieved through the use of redundancy—the failure of a component is offset by the use of a redundant component to achieve the same mission or purpose.

The introduction of new technology, especially digital technology, and the increasing complexity of system designs (most of it made possible by the use of computers) is starting to produce an important change in the nature of accidents. While accidents related to hardware failure are being reduced, *system accidents* (a term coined by Perrow [Per84]), are increasing in importance.

System accidents arise in the interactions *among* components (electromechanical, digital, and human) rather than from the failure of individual components. In these accidents, the components may all operate according to their specification, that is, there is no "failure," but unexpected or desired interactions among components leads to the loss. For example, recently the FAA issued an Advisory Directive (AD) related to an accident where the stall warning on a particular aircraft can be delayed if the stall conditions occur while the flaps are moving. In this case (as is true for most system accidents), each component worked according to its specification (i.e., the stall warning software and the control surface software), but subtle component interactions at the system level allowed the stall warning to be delayed.

System accidents in general are related to (1) interactive complexity and (2) tight coupling.

The underlying factor in interactive complexity 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 interactively 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.

For component failure accidents, the risk involved in the use of COTS products can often be handled through special system design features. All components can potentially fail and software is no different in this respect, where software failure is defined as not satisfying its specified required behavior. Traditional system design techniques are usually applicable whether the component providing the required functionality is analog or digital. The only difference, from a system design perspective, is that the failure modes may differ. The unique failure modes for software may or may not make dealing with them practical at the system design level, but all such failure modes need to be considered. Digital systems commonly have a larger possible set of failure modes than analog systems and this set may or may not be able to be handled effectively, depending on the particular system.

System accidents present a greater challenge and need to be considered in more detail.

2.2 System Safety

System safety is the part of system engineering that deals with the challenge of preventing or reducing both component failure and system accidents in complex systems. It was developed as a response to 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. It became the rationale

behind the U.S. military standard Mil-Std-882 and its various versions over the past 30 years.

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 system 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 (i.e., the uncommanded steering behavior) 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. And in the context of this meeting, the information necessary to perform system hazard analysis may not be available for COTS components.

2.3 The Role of Software in System 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 industries (such as automobiles) where operators are not as 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. This is the approach embodied in the new IEC 61508 standard. 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 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.

This fact has very important implications for COTS and risk. Safety is a system property—not a component property. Using a COTS product, even if it has very high reliability (or SIL level), does not imply that the product is safe when it interacts with other system components. The problem is exacerbated with software because software usually controls many if not all of the interactions between system components. Techniques for dealing with COTS by simply equating software reliability or correctness (consistency with specifications) will not prevent system accidents. To understand what *is* necessary, some background about how system safety deals with software is required.

2.4 System Safety and Software

Because of the unique features of digital systems, and software in particular, traditional system safety approaches must be extended to deal with systems having digital components, in particular, to handle the new levels of complexity, new types of failure modes, and new types of problems arising in the interaction between components in computer-intensive system designs [Lev95].

In the system safety approach to building systems containing safety-critical software, 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 approach 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 black-box 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 (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.

With this methodology, the information provided by system engineers about potential system hazards is used to build safety into the software starting from the earliest tasks and continuing throughout the software development and evolution process. As an example, consider a simple aircraft altitude switch that issues a command to another device (perhaps to turn it on) when the aircraft descends below a threshold altitude. Assume that the altitude is determined by the use of multiple altimeters on board the aircraft, each of which sends the altitude it senses to the altitude switch software. When the output command of the altitude switch is potentially hazard-reducing (e.g., the pilot is being alerted that the aircraft has descended below its minimum safe altitude), the safest design will be to issue the output command if any one of the altimeters indicates the threshold altitude has been crossed. Alternatively, if the output is hazard-increasing (e.g., the information is used to release a missile), the safest software functional design might be to require agreement by all the altimeters before issuing the command.

Note that the safety of the design of the component (the altitude switch) is dependent on the use of the output from the switch within the overall system design. The particular system hazard being eliminated or mitigated will determine whether any particular altitude switch behavior is safe or not.

3 Implications for Systems using COTS Software

All COTS software is not alike, and we first need to differentiate between *application software* that provides a direct system function and *system software* that provides a platform on which application software is executed. Each of these provides unique challenges for system safety and COTS.

3.1 System Software

System software has almost always been specially constructed for very dangerous systems in order to allow the thorough analysis and high confidence required. The overall goals for the two main types of system software are at polar opposites:

- **Platforms for hobbyists, business productivity tools, and home use**: Be first to market with the most features and assume that consumers are more interested in features than reliability or quality
- **Platforms for critical, real-time systems**: Provide the minimum features required to do the basic task in such a way that high confidence can be assured.

The shrinkwrap system software industry assumes that users will adapt to any problems with the software and that limitations of liability (through waivers or through laws limiting liability) will indemnify them against any damages or legal action resulting from the use of their product.

In this market atmosphere and because of the difficulty—indeed usually impossibility—of providing high assurance of acceptable behavior for a product that has not been designed to be amenable to such assurance, the only feasible approach is to buy and use system software that has been specially constructed for critical systems. The key here is simplicity and features limited to those that are needed and can be verified. Such system software exists and is used for critical avionics and other systems today. Because the market for such system software is limited, we need to ensure that these products and companies survive. Any additional costs involved in using limited-market products will be well made up for by the reduction in analysis required to ensure safety.

Many of the non-technical benefits of COTS, such as distributing costs over a larger user base, are preserved by the use of these special products although the savings

may be reduced in the short term by a smaller marketplace. The long-term savings, which need to include such factors as the costs involved in accidents, should actually increase.

There is one special case with a somewhat less pessimistic conclusion: The use of COTS system software with acceptable risk may be feasible if the system can be protected from any potential unsafe behavior of the COTS component as a whole. In this case, system software is handled in much the same way that application software is treated, as described in the next section.

3.2 Application Software

The use of COTS application software or specialized components, such as device drivers or GPS receivers, is more feasible but still presents difficulties and has some prerequisites to achieving acceptable risk.

As discussed above, safety is a system property, not a component property. It is not possible to determine for a component in isolation if its use will be safe for any system in which it might be employed, as illustrated by the altitude switch example earlier. Each system design requires a complete safety analysis of the interaction of the components and the hazards that may arise from such an interaction. Therefore, a system that contains COTS components will need to undergo a complete system hazard analysis. The question is what information is required to accomplish that and can that information be obtained for digital COTS products.

If the component or function is not critical from a system safety standpoint, i.e., it can be shown that any behavior of that component cannot affect a system hazard, then COTS presents no problems. The only issue becomes one of being able to make that determination at the system level.

If the component or function is critical but system engineers are able to create a system design in which any failure or untoward behavior of that component is mitigated, i.e., cannot contribute to a hazard, then again the problem is relatively simple and solvable using standard system engineering techniques such as redundancy, monitoring, fault tolerance, or new forms of these techniques such as wrappers. The analysis involved is nontrivial but no different than that required for non-COTS components.

The most serious problems arise when the COTS component is performing a critical function (with respect to safety) and adequate protection against any potential behavior of this component cannot be provided in the overall system design. In this case, those performing the system hazard analysis must have adequate information about the potential behavior of the component to be able to determine whether its use within the system being designed provides acceptable safety. For hardware components, that information can either be provided by observation or by testing and analysis of observable features. For software, such observation is not possible and execution or testing under diverse conditions does not suffice to provide enough information due to the enormous state space involved (e.g., our model of the blackbox behavior of TCAS II, an airborne collision avoidance system, contains 10^40 states).

Let's consider what information might suffice for a system hazard analysis of COTS software. If, as is argued above, almost all software-related accidents involve requirements flaws and the system hazard analysis is concerned only with the external behavior of the component, a blackbox specification of component behavior is all that is theoretically required. The internal design of the component is irrelevant as long as that internal design results in the blackbox requirements being satisfied. Therefore, a blackbox behavioral specification of the COTS component's behavior will allow the system safety engineer to determine whether the use of that component will have acceptable risk. This blackbox specification must specify all important or relevant externally visible behavior including timing.

The next question is whether such a specification is possible. For the past 12 years, my students and I have been identifying completeness criteria for blackbox software requirements specifications [JLH91, Lev95] creating specification languages that enforce completeness [LHR99, Lev00], and demonstrating the feasibility of producing such specifications on complex systems [LHH94, RE97, LRZ00]. The cost of producing such specifications, however, is non-trivial. Much more important, COTS producers must be willing to provide such specifications. And analysis procedures and tools are required that can navigate and evaluate the enormous state spaces that may be involved. Many such tools already exist and more are being created within research labs such as my own.

Theoretically, blackbox software behavior specifications are obtainable through testing and observation by any purchaser of the box since the behavior is, by definition, externally visible. However, the cost (and even feasibility) of the purchaser creating such specifications

through testing and observation makes this impractical. Therefore, the developers and sellers of the software must be willing to provide the information, which is essentially what should be provided in an adequate user manual. No proprietary information about the software need be divulged—only externally observable behavior.

Practically, because many (if not most) software developers never create such specifications (although they should), providing them would require extra effort on their part. If the market is small or the profit margins small, they may be unwilling to make this effort. If this case, the only hope of obtaining the information necessary to perform an adequate system hazard analysis is to pay for it. Again, this reduces any potential savings from COTS products. However, the cost of producing such specifications can theoretically be shared by all customers of the component, and the producers will have a market advantage over their competitors.

4 Conclusions

In many cases, using COTS components in safety-critical systems with acceptable risk will simply be infeasible. In these cases, it will be cheaper and safer to provide special-purpose software—using COTS amounts to false economy that will cost more in the end. There are, however, situations in which COTS components can be assured to have adequate system safety. In these cases, either the system design must allow protection against any possible hazardous software behavior or a complete blackbox behavior specification must be provided by the producer of that component in order to perform a system hazard analysis. For complex software, special system hazard analysis techniques and tools may be needed to assist the system engineer in this task.

References

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

[JLH91] Jaffe, M.S., Leveson, N.G., Heimdahl, M.P.E., and Melhart, B. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

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

[Lev00] Leveson, N.G.. Completeness in Formal Specification Language Design for Process-Control Systems. Technical Report, Software Engineering Research Lab, MIT, 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.

[LHH94] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H. and Reese, J.D. Requirements specification for process-control systems. *IEEE Transations on Software Engineering*, SE-20(9), September 1994.

[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.

[LRZ00] Leveson, N., Rodriguez, M. and Zimmerman, M. Specification of the Vertical Flight Control Logic for a Civil Transport Aircraft. *Digital Aviation Systems Conference*, October 2000 (to appear).

[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.

[RE97] Modugno, F., Leveson, N.G., Reese, J.D., Partridge, K., and Sandys, S.D. Experimental application of safety analysis to a software requirements specification. *Requirements Engineering Journal*, 1997.

[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.