The Role of Software in Recent Aerospace Accidents*

Nancy G. Leveson, Ph.D.; Aeronautics and Astronautics Department,
Massachusetts Institute of Technology; Cambridge MA
leveson@mit.edu and http://sunnyday.mit.edu
Keywords: software safety

Abstract

This paper describes causal factors related to software that are common to several recent spacecraft accidents and what might be done to mitigate them.

Introduction

In the process of a research project to evaluate accident models, I looked in detail at a variety of spacecraft and aircraft accidents that in some way involved software [8]. The accidents studied all demonstrated the usual cultural, organizational, and communications problems such as complacency, diffusion of or lack of responsibility and authority for safety, low-level status and inappropriate organizational placement of the safety program, and limited communication channels and poor information flow. These typical problems are well known and the solutions clear although sometimes difficult to implement. Software contributions to accidents are less well understood, however.

The accidents investigated were the explosion of the Ariane 5 launcher on its maiden flight in 1996; the loss of the Mars Climate Orbiter in 1999; the destruction of the Mars Polar Lander sometime during the entry, deployment, and landing phase in the following year; the placing of a Milstar satellite in an incorrect and unusable orbit by the Titan IV B-32/Centaur launch in 1999; the flight of an American Airlines B-757 into a mountain near Cali, Columbia; the collision of a Lufthansa A320 with an earthbank at the end of the runway at Warsaw, and the crash of a China Airlines A320 short of the runway at Nagoya, Japan.

On the surface, the events and conditions involved in the accidents appear to be very different. A more careful, detailed analysis of the systemic factors, however, reveals striking similarities. Only the root causes or systemic factors are considered here, that is, the causal factors that allowed the specific events to occur and that affect general classes of accidents. In the Challenger accident, for example, the specific even leading to the loss was the O-ring failure, but the systemic factors included such things as flawed decision making, poor problem reporting, lack of trend analysis, a "silent" or ineffective safety program, communication problems, etc.

Overconfidence and Overreliance on Digital Automation: All the accidents involved systems built within an engineering culture that had unrealistic expectations about software and the use of computers. For example, the official Ariane 5 accident report notes that software was assumed to be correct until it was shown to be faulty. The opposite assumption is more realistic.

Engineers often underestimate the complexity of software and overestimate the effectiveness of testing. It is common to see risk assessments that assume testing will remove all risk associated with digital components. This form of complacency plays a part in the common proliferation of software functionality and in unnecessary design complexity.

In the aircraft accidents examined, overconfidence in automation both (1) encouraged engineers to trust software over humans and give final authority to the computer rather than the pilot, and (2) encouraged pilots to trust their computer-based decision aids beyond the point where they should have.

Some of the technical inadequacies in high-tech aircraft system design stem from lack of confidence in the human and overconfidence in the automation. In several of the Airbus accidents, the pilots found themselves fighting the automation for control of the aircraft---which

* This research was partially supported by a grant from the NASA Ames Design for Safety program and by the NASA IV&V Center Software Initiative program.

had been designed to give ultimate authority to the automation.

Even if automation is considered to be more reliable than humans, it may be a mistake not to allow flexibility in the system for emergencies and allowance for pilots to override physical interlocks, such as the inability of the pilots to operate the ground spoilers and engine thrust reversers in the Warsaw A-320 accident because the computers did not think the airplane was on the ground. Reliable operation of the automation is not the problem here; the automation was very reliable in all these cases. Instead the issue is whether software can be constructed that will exhibit correct appropriate behavior under *every* foreseeable and unforseeable situation and whether we should be trusting software over pilots.

At the same time, some of the aircraft accident reports cited the lack of automated protection against or nonalerting of the pilots to unsafe states, such as out-of-trim situations. A sophisticated hazard analysis and close cooperation among system safety engineers, human factors engineers, aerospace engineers, and software engineers is needed to make these difficult decisions about task allocation and feedback requirements.

Engineers are not alone in placing undeserved reliance on software. Research has shown that operators of highly reliable automated systems (such as flight management systems) will increase their use of and reliance on automation as their trust in the system increases. At the same time, merely informing flightcrews of the hazards of overreliance on automation and advising them to turn it off when it becomes confusing is insufficient and may not affect pilot procedures when it is most needed.

The European Joint Aviation Authorities' Future Aviation Safety Team has identified "crew reliance on automation" as the top potential safety risk in future aircraft [5]. This reliance on and overconfidence in software is a legitimate and important concern for system safety engineering.

Not Understanding the Risks Associated with Software: The accident reports all exhibited the common belief that the same techniques used for electromechanical components will work in software-intensive systems. However, the failure modes for software are very different than those for physical devices and the contribution of software to accidents is also different: Engineering activities must be changed to reflect these differences. Almost all software-related accidents can be traced back to flaws in the requirements specification and not to coding errors. In these cases, the software performed exactly as specified (the implementation was "correct") but the specification was incorrect because (1) the requirements were incomplete or contained incorrect assumptions about the required operation of the system components being controlled by the software or about the required operation of the computer or (2) there were unhandled controlled-system states and environmental conditions. This in turn implies that the majority of the software system safety effort should be devoted to requirements analysis, including completeness (we have specified an extensive set of completeness criteria), correctness, potential contribution to system hazards, robustness, and possible operator mode confusion and other operator errors created or worsened by software design.

Confusing Reliability and Safety: Accidents are changing their nature. We are starting to see an increase in *system accidents* that result from dysfunctional interactions among components, not from individual component failure. Each of the components may actually have operated according to its specification (as is true for most software involved in accidents), but the combined behavior led to a hazardous system state. When humans are involved, often their behavior can only be labeled as erroneous in hindsight–at the time and given the context, their behavior was reasonable (although this does not seem to deter accident investigators from placing all or most of the blame on the operators).

System accidents are caused by interactive complexity and tight coupling. Software allows us to build systems with a level of complexity and coupling that is beyond our ability to control; in fact, we are building systems where the interactions among the components cannot be planned, understood, anticipated, or guarded against. This change is not solely the result of using digital components, but it is made possible because of the flexibility of software.

Standards for commercial aircraft certification, even relatively new ones, focus on component reliability and redundancy and thus are not

effective against system accidents. In the aircraft accidents studied, the software satisfied its specifications and did not "fail" yet the automation obviously contributed to the flight crews' actions and inactions. Spacecraft engineering in most cases also focuses primary effort on preventing accidents by eliminating component failures or preparing for failures by using redundancy. These approaches are fine for electromechanical systems and components, but will not be effective for software-related accidents.

The first step in handling system accidents is for engineers to recognize the need for change and to understand that safety and reliability are *different* qualities for software---one does not imply the other. One of the founders of system safety, C.O. Miller, cautioned that "distinguishing hazards from failures is implicit in understanding the difference between safety and reliability" [13]. Although confusing reliability with safety is common in engineering (and particularly common in software engineering), it is perhaps most unfortunate with regard to software as it encourages spending much of the effort devoted to safety on activities that are likely to have little or no effect.

Overrelying on Redundancy: Redundancy usually has a greater impact on reliability than safety. System accidents, for example, will not be decreased at all by the use of redundancy. In fact, the added complexity introduced by redundancy has frequently resulted in accidents. In addition, redundancy is most effective against random wearout failures and least effective against requirements and design errors–the latter being the only type found in software. For example, the Ariane report notes that according to the culture of the Ariane program, only random failures are addressed and they are primarily handled with redundancy. This approach obviously failed when on the Ariane 5's first flight both Inertial Reference System computers shut themselves down (exactly as they were designed to do) as a result of the same unexpected input value.

To cope with software design errors, "diversity" has been suggested in the form of independent groups writing multiple versions of software with majority voting on the outputs. This approach is based on the assumption that such versions will fail in a statistically independent manner, but this assumption has been shown to be false in practice and by scientific experiments (see, for example, [4]). Common-cause (but usually different) logic errors tend to lead to incorrect results when the various versions attempt to handle the same unusual or difficult-to-handle inputs. In addition, such designs usually involve adding to system complexity, which can result in failures itself. A NASA study of an experimental aircraft with two versions of the control system found that all of the software problems occurring during flight testing resulted from errors in the redundancy management system and not in the control software itself, which worked perfectly [12].

Assuming Risk Decreases over Time: In the Milstar satellite loss, the Titan Program Office had decided that because the software was "mature, stable, and had not experienced problems in the past," they could use the limited resources available after the initial development effort to address hardware issues. Other accidents studied had this same flawed approach to resource prioritization.

A common assumption is that risk decreases over time as accident-free operation accumulates. In fact, risk usually increases over time, particularly in software-intensive systems. The Therac-25, a radiation therapy machine that massively overdosed five patients due to software flaws, operated safely thousands of times before the first accident. Industrial robots operated safely around the world for several million hours before the first fatality.

Risk may increase over time because caution wanes and safety margins are cut, because time increases the probability the unusual conditions will occur that trigger an accident, or because the system itself or its environment changes. In some cases, the introduction of an automated device may actually change the environment in ways not predicted during system design.

Software also tends to be frequently changed and "evolves" over time, but changing software without introducing errors or undesired behavior is much more difficult than building correct software in the first place. The more changes that are made to software, the more "brittle" the software becomes and the more difficult it is to make changes without introducing errors.

The history of accidents shows that a strong system safety program is needed during

operations. All changes to the software must be analyzed for their impact on safety. Such change analysis will not be feasible unless special steps are taken during development to document the information needed. Incident and accident analysis, as for any system, will also be important as well as performance monitoring and periodic operational process audits.

The environment in which the system and software are operating will change over time, partially as a result of the introduction of the automation or system itself. Basic assumptions made in the original hazard analysis process must have been recorded and then should be periodically evaluated to ensure they are not being violated in practice. For example, in order not to distract pilots during critical phases of flight, TCAS includes the ability for the pilot to switch to a Traffic-Advisory-Only mode where traffic advisories are displayed but display of resolution advisories (escape maneuvers) is inhibited. It was assumed in the original TCAS system design and hazard analysis that this feature would be used only during final approach to parallel runways when two aircraft come close to each other and TCAS would call for an evasive maneuver. The actual use of this feature in practice would be an important assumption to check periodically to make sure it is not being used in other situations where it might lead to a hazard. But that requires that the assumption was recorded and not forgotten.

Ignoring Warning Signs: Warning signs almost always occur before major accidents. For example, all the aircraft accidents considered in this research had had precursors but priority was not placed on fixing the causal factors before they reoccurred or the responses were inadequate to prevent future loss. Two of the three aircraft accidents studied involved problems for which software fixes had been created but for various reasons had not been installed on the specific aircraft involved. The reasons for this omission are complicated and sometimes involved politics and marketing (and their combination) as much as complacency or cost factors.

Engineers noticed the problems with the Titan/Centaur software after it was delivered to the launch site, but nobody seemed to take them seriously. The problems experienced with the Mars Climate Orbiter software during the early stages of the flight did not seem to raise any red flags. The system safety information system should include the collection of such information and its analysis to detect problems before they cause serious losses.

Inadequate Cognitive Engineering: Commercial aviation is the first industry where shared control of safety-critical functions between humans and computers has been widely implemented. The very difficult problems that result, such as those associated with mode confusion and deficiencies in situational awareness, are slow to be recognized and acknowledged. It is more common to simply blame the pilot for the accident than to investigate the aspects of system design that may have led to the human error(s).

All the aircraft accident reports focused on pilot error. Some of the spacecraft accidents also focused their investigations on the ground controllers and why they did not catch the software problems before the loss instead of focusing on why the software problems were introduced in the first place and not caught before operational deployment of the system.

Cognitive engineering, particularly that directed at the influence of software design on human error, is still in its early stages. Human factors experts have written extensively on the potential risks introduced by the automation capabilities of glass cockpit aircraft. Among those identified are: over reliance on automation; shifting workload by increasing it during periods of already high workload and decreasing it during periods of already low workload; being "clumsy" or difficult to use; being opaque or difficult to understand; and requiring excessive experience to gain proficiency in its use. In the Cali accident, for example, the accident report noted task saturation and overload, poor situation awareness (inadequate mental models of the automation and the situation), and distraction from appropriate behavior.

Researchers have suggested that pilots of high-tech aircraft can lose awareness of the current aircraft flight mode or exhibit other forms of mode confusion. In addition, many of the problems found in human--automation interaction lie in the human not getting adequate feedback to monitor the automation and to make appropriate decisions.

System safety needs to consider these potential problems in any hazard analysis. Just as hazard

analysis needs to begin in the early conceptual stages, so does the design of the human--computer interaction. We have defined a system engineering process that is both human-centered and safety-driven [11]. The information generated by system safety engineers in the system hazard analysis process can be extremely useful in defining operator goals and responsibilities, task allocation principles, and operator task and training requirements, i.e., in the activities involved in designing safer human--computer interactions.

Inadequate Specifications: The Mars Polar Lander accident report notes that the system-level requirements document did not specifically state the failure modes the requirement was protecting against (in this case, possible sensor transients) and speculates that the software designers or one of the reviewers might have discovered the missing software requirement if they had been aware of the rationale underlying the system requirements. The Ariane accident report refers in several places to inadequate specification practices and notes that the structure of the documentation obscured the ability to review the critical design decisions and their underlying rationale.

The small part of the Mars Polar Lander software requirements specification shown in the accident report (which may very well be misleading) avoids all mention of what the software must *not* do. In fact, some standards and industry practices even forbid such negative requirements statements. The result is that software specifications often describe nominal behavior well but are very incomplete with respect to required software behavior under off-nominal conditions and rarely describe what the software is not supposed to do. Most safety-related requirements and design constraints are best described using such negative requirements or design constraints so they are often omitted.

This is a place where good system hazard analysis can be very helpful. Unfortunately, many hazard analyses treat software superficially at best. The hazard analysis produced for the Mars Polar Lander (MPL) during the accident investigation is typical. The JPL report on the MPL loss identifies the hazards for each phase of the entry, descent, and landing sequence, such as (1) *Propellant Line Rupture*, (2) *Excessive horizontal velocity causes lander to tip over at touchdown*, and (3) *Premature shutdown of the descent engines*. For software, however, only one statement---*Flight software fails to execute properly*---is identified, and it is labeled as common to all phases.

The problem with such vacuous statements is that they provide no useful information---it is equivalent to simply substituting a single statement for all the other identified system hazards with *Hardware fails to operate properly*. Singling out the JPL engineers is unfair here because I find the same types of useless statements about software in almost all the fault trees and other hazard analyses I see in industry. Boxes in fault trees that simply say *Software Fails* can be worse than useless because they are untrue---all software misbehavior will not cause the identified hazard---and it leads to nonsensical activities like using a general reliability figure for software (assuming one believes such a number can be produced) in quantitative fault tree analyses when it does not reflect in any way the probability of the software exhibiting a particular hazardous behavior.

Instead, specific hazardous software behavior needs to be identified. In a collision avoidance system, for example, a fault tree box might contain: (1) *Collision avoidance logic calls for a reversal of an advisory when the pilot has insufficient time to respond* or (2) two boxes connected by an AND might contain *Software issues a crossing advisory* and *The pilot does not follow his/her advisory*. These two identified hazardous software behaviors might be translated into two system design constraints:

[1.] *The software must not call for a reversal of an advisory when two aircraft are separated by less than 200 feet vertically and 10 seconds or less remain to closest point of approach and*
[2.] *Crossing maneuvers must be avoided if possible*

along with a pilot procedural requirement to follow all advisories and to continue to do so until the advisory is removed.

Complete and understandable specifications are not only necessary for development, but they are critical for operations and the handoff between developers, maintainers, and operators. The ground operations staff in the spacecraft accidents and the pilots in the aircraft accidents all had misunderstandings about the automation and how to use it. All three aircraft accident

reports cited inadequate, conflicting, or poorly designed documentation and information presentation.

A specification method, called Intent Specifications [7], has been defined that integrates safety information into the engineering decision-making environment during the early stages of system conceptual design and functional allocation, encourages documentation of design rationale and safety-related design assumptions, and provides complete traceability from high-level requirements and hazard analyses down through the levels of system design to the component implementation details and vice versa. While intent specifications are useful during the original system design, they should be particularly helpful during operations in performing safety assessments on potential changes to the system and software.

Flawed Review Process: In general, software is difficult to review and the success of such an effort is greatly dependent on the quality of the specifications. However, identifying unsafe behavior, i.e., the things that the software should not do and concentrating on that behavior for at least part of the review process, helps to focus the review and to ensure that critical issues are adequately considered. The fact that specifications usually include only what the software should do and omit what it should not do makes this type of review even more important and effective in finding serious problems. Such unsafe (or mission-critical) behavior should be identified in the system engineering process before software development begins. The design rationale and design features used to prevent the unsafe behavior should also have been documented and can be the focus of such a review. This presupposes, of course, a system safety process to provide the information, which does not appear to have existed for the projects that were involved in the accidents studied.

The two identified Mars Polar Lander software errors, for example, involved incomplete handling of software states and are both examples of very common specification flaws and logic omissions often involved in accidents. As such, they were potentially detectable by formal and informal analysis and review techniques. The Ariane report also says that the limitations of the inertial reference system software were not fully analyzed in reviews, and it was not realized that the test coverage was inadequate to expose such limitations.

Software hazard analysis and requirements analysis techniques exist (and more should be developed) to detect all these types of incompleteness. To make such a review feasible, the requirements should include only the externally visible (blackbox) behavior; all implementation-specific information should be put into a separate software design specification (which will be subject to a later software design review by a more limited set of reviewers). The only information relevant to requirements review at this level is the software behavior that is visible outside the computer. Specifying only blackbox behavior (in engineering terms, this is often referred to as the *transfer function* across the digital component) allows a wide set of reviewers to concentrate on the information of importance to them without being overwhelmed by internal design information that has no impact on externally observable behavior.

The language used to specify the software requirements is critical to the success of the review. The best way to find errors in the software requirements is to include a large range of disciplines and expertise in the review process, including system safety engineers. Formal specification methods have tremendous potential for enhancing our ability to provide correct and complete requirements. In addition, executable specification can be helpful in understanding the implications of complex software behavior. We showed in our TCAS project that it is possible for a formal, executable requirements specification to be readable and understandable with a minimum of training and without requiring advanced degrees in discrete math and logic [10], but most designers of formal requirements specification languages have not put a high priority on readability and learnability. We have used what we learned during the TCAS project to design an even more reviewable new requirements specification language.

Inadequate System Safety Engineering: All of the accident reports studied are surprisingly silent about the safety programs involved. One would think that the safety activities and why they had been ineffective would figure prominently in the investigations, assuming, of course, there were active safety programs and the efforts were not marginalized or ignored by the

other engineering activities. Judging only from the information (or lack of it) provided in the accident reports, it is likely that none of these projects had a robust system safety or software system safety program.

Providing the information needed to make safety-related engineering decisions is the major contribution of system safety techniques to engineering. It has been estimated that 70-90% of the safety-related decisions in an engineering project are made during the early concept development stage [2]. When hazard analyses are not performed, are done only after the fact (for example, as a part of quality or mission assurance of a completed design), or are performed but the information is never integrated into the system design environment, they can have no effect on these decisions and the safety effort reduces to a cosmetic and perfunctory role.

The Titan accident provides an example of what happens when such analysis is not done. The risk analysis, in that case, was not based on determining the steps critical to mission success but instead considered only the problems that had occurred in previous launches. Software constant generation (an important factor in the Milstar satellite loss) was considered to be low risk because there had been no previous problems with it. There is, however, a potentially enormous (perhaps unlimited) number of errors related to software and considering only those mistakes made previously, while certainly prudent, is not adequate. Proper hazard analysis that examines all the ways the system components (including software) or their interaction can contribute to accidents needs to be performed and used in decision making.

The Mars Climate Orbiter accident report recommended that the NASA Mars Program institute a classic system safety engineering program, i.e., continually performing the system hazard analyses necessary to explicitly identify mission risks and communicating these risks to all segments of the project team and institutional management; vigorously working to make tradeoff decisions that mitigate the risks in order to maximize the likelihood of mission success; and regularly communicating the progress of the risk mitigation plans and tradeoffs to project, program, and institutional management. The other spacecraft accident reports, in contrast, recommended applying classic reliability engineering approaches that are unlikely to be effective for system accidents or software-related causal factors.

Violation of Basic Safety Engineering Practices in the Digital Parts of the System: Although system safety engineering textbooks and standards include principles for safe design, software engineers are almost never taught them. As a result, software often does not incorporate basic safe design principles---for example, separating and isolating critical functions, eliminating unnecessary functionality, designing error-reporting messages such that they cannot be confused with critical data (see the Ariane 5 loss), and reasonableness checking of inputs and internal states.

Consider the Mars Polar Lander loss as an example. The JPL report on the accident states that the software designers did not include any mechanisms to protect against transient sensor signals nor did they think they had to test for transient conditions. They also apparently did not include a check of the current altitude before turning off the descent engines. Runtime reasonableness and other types of checks should be part of the design criteria used for any real-time software.

Another basic design principle for mission-critical software is that unnecessary code or functions should be eliminated or separated from mission-critical code and its processing. The Arian 5 and Titan IVB-32 accidents involved code that was not needed (it was in some reused software designed for other spacecraft). In the case of Mars Polar Lander, the code that caused the problems was necessary (in fact, it was critical) but was executing at a time when it was not needed. I am sure that each of these decisions was considered carefully, but the tradeoffs may not have been made in an optimal way and risk may have been discounted with respect to other properties.

Software Reuse without Appropriate Safety Analysis: It is widely believed that because software has executed safely in other applications, it will be safe in the new one. This misconception arises from confusion between software reliability and safety. As stated, most accidents involve software that is doing exactly what it was designed to do, but the designers misunderstood what behavior was required and would be safe.

The blackbox (externally visible) behavior of a component can only be determined to be safe by analyzing its effects on the system in which it will be operating, that is, by considering the specific operational context. The fact that software has been used safely in another environment provides *no* information about its safety in the current one. In fact, reused software is probably less safe because the original decisions about the required software behavior were made for a different system design and were based on different environmental assumptions. *Changing the environment in which the software operates makes all previous usage experience with the software irrelevant for determining safety*.

A reasonable conclusion to be drawn is not that software cannot be reused, but that a safety analysis of its operation in the new system context is mandatory: Testing alone is not adequate to accomplish this goal. For complex designs, the safety analysis required stretches the limits of current technology. For such analysis to be technically and financially feasible, reused software must contain only the features necessary to perform critical functions: Both the Ariane 5 and the Titan software contained unnecessary functions that led to the losses and the MPL loss involved a necessary function operating when it was not necessary.

COTS software is often constructed with as many features as possible to make it commercially useful in a variety of systems. Thus there is tension between using COTS versus being able to perform a safety analysis and have confidence in the safety of the system. This tension must be resolved in management decisions about risk---ignoring it only leads to accidents and potential losses that are greater than the additional cost of designing and building new components instead of buying them.

If software reuse is to result in acceptable risk, then system and software modeling and analysis techniques must be used to perform the necessary safety analyses. This process is not easy or cheap. Introducing computers does not preclude the need for good engineering practices, and it almost always involves higher costs despite the common myth that introducing automation will save money. Our blackbox formal requirements specification language contains the information necessary for such a safety analysis and therefore should be useful not only in the original system development but when the software is to be reused.

Unnecessary Complexity and Software Functions: One of the most basic concepts in engineering critical systems is to "keep it simple." The seemingly unlimited ability of software to implement desirable features often, as in the case of most of the accidents examined in this paper, pushes this basic principle into the background: Creeping featurism is a common problem in software and engineering. As stated earlier, the Ariane and Titan accidents involved software functions that were not needed, but surprisingly the decision to put in or to keep (in the case of reuse) these unneeded features was not questioned in the accident reports. The Mars Polar Lander accident involved software that was executing when it was not necessary to execute. In the case of the Titan IVB-32, the report explains that the software roll rate filter involved in the loss of the Milstar satellite was not needed but was kept in for "consistency." The exact same words are used for software functions leading to the loss of the Ariane 5. Neither report explains why consistency was assigned such high priority. In all these projects, tradeoffs were obviously not considered adequately, perhaps partially due to complacency about software risk.

The more features included in software and the greater the resulting complexity (both software complexity and system complexity), the harder and more expensive it is to test, to provide assurance through reviews and analysis, to maintain, and to reuse in the future. Engineers need to start making these hard decisions about functionality with a realistic appreciation of their effect on development cost and eventual system safety and system reliability.

Operational Personnel Not Understanding the Automation: Neither the MPL nor the Titan mission operations personnel understood the system or software well enough to interpret the data they saw as indicating there was a problem in time to prevent the loss. Complexity in the automation combined with poor documentation and training procedures are contributing to this problem, which is becoming a common factor in aircraft accidents. Accidents, surveys, and simulator studies have emphasized the problems pilots are having in understanding digital

automation and have shown that pilots are surprisingly uninformed about how the automation works [14], [11].

Accidents have further demonstrated that proficiency in the use of sophisticated automation, such as a FMS (Flight Management System), without adequate knowledge about the logic underlying critical features, such as the design and programmed priorities of its navigation database or autopilot override functions, can lead to its misuse and to accidents. Problems are especially found with controls and operations the crews rarely experience in daily flight, such as unusual mode changes and manual overrides.

Either the design of the automation we are building needs to be simplified so it is understandable or new training methods are needed for those who must deal with the clumsy, unpredictable, and inconsistent automation we are designing, or both.

Test and Simulation Environments that do not Match the Operational Environment: It is always dangerous to conclude that poor testing was the "cause" of an accident. After the fact, it is always easy to find a test case that would have uncovered a known error, but it is usually difficult to prove that the particular test case would have been selected beforehand, even if testing procedures were changed. By definition, the cause of an accident can always be stated as a failure to test for the condition that was determined, after the accident, to have led to the loss. However, in the accidents studied, there do seem to be omissions that reflect poor decisions related to testing, particular with respect to the accuracy of the simulated operational environment.

A general principle in testing aerospace systems is to *fly what you test and test what you fly*. This principle was violated in all the spacecraft accidents, especially with respect to software. The software test and simulation processes must reflect the environment accurately. Although implementing this principle is often difficult or even impossible for spacecraft, no reasonable explanation was presented in the reports for some of the omissions in the testing for these systems. An example was the use of Ariane 4 trajectory data in the specifications and simulations of the Ariane 5 software even though the Ariane 5 trajectory was known to be different. Another example was not testing the Titan/Centaur software with the actual load tape prior to launch.

Deficiencies in Safety-Related Information Collection and Use: In all the spacecraft accidents, the existing formal anomaly reporting system was bypassed (in Ariane 5, there is no information about whether one existed) and informal email and voice mail was substituted. The problem is clear but not the cause, which was not included in the reports and perhaps not investigated. When a structured process exists and is not used, there is usually a reason. Some possible explanations may be that the system is difficult or unwieldy to use or it involves too much overhead. Such systems may not be changing as new technology changes the way engineers work.

There is no reason why reporting something within the problem-reporting system should be much more cumbersome than adding an additional recipient to the email. The Raytheon CAATS (Canadian Automated Air Traffic System) project implemented an informal email process for reporting anomalies and safety concerns or issues that reportedly was highly successful [3]. New hazards and concerns will be identified throughout the development process and into operations, and there must be a simple and non-onerous way for software engineers and operational personnel to raise concerns and safety issues and get questions answered at any time.

Conclusion: The incidence of system accidents is increasing as engineering designs rely more and more on software. But system accidents are exactly the type of accident that system safety was invented to handle 50 years ago, and it should be very effective against the system accidents stemming from misunderstood software requirements and the dysfunctional system interactions typical of software-related accidents. This approach does not require that system components exhibit ultra-high reliability, only that a set of specific behaviors do not occur. For software, this distinction is critical: It is much easier to design software to prevent particular behaviors than to guarantee that it will always do the "right" thing. In fact, the latter may be impossible. Classic reliability engineering techniques, such as failure analysis and redundancy, will be less important for software

components than for electromechanical components.

I have noticed, however, that over the years system safety engineering has increasingly drifted toward using reliability engineering techniques and away from classic system safety approaches and has, in particular, adopted this approach for software. This trend may simply be a result of lack of knowledge about software or it may reflect a lack of appropriate tools to assist in applying system safety approaches to software. A further influence may be that computer science has always been concerned with computer reliability and has focused almost exclusively on this quality. Only recently has safety become an issue. Therefore, almost all existing software engineering techniques focus on software reliability, i.e., assuring that the software correctly or reliably satisfies the specified requirements (which may be incomplete, incorrect, or unsafe).

We have created demonstration projects to show how classic system safety approaches can be applied to software-intensive systems (see, for example, [9] and [11]). In addition, the MIT Software Engineering Research Laboratory (SERL) is working to create new techniques and tools to support software system safety analysis and design.

References

1. Australia. 1996. Bureau of Air Safety Investigation, Department of Transport and Regional Development. *Advanced Technology Aircraft Safety Survey Report*. June.

2. Johnson, W. G. 1980. *MORT Safety Assurance Systems*. New York: Marcel Dekker, Inc.

3. Joyce, Jeffrey. Conversation w/author, 2001.

4. Knight, J.C., and Nancy G. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Transactions on Software Engineering* SE-12, no. 1 (January 1986): 96-109.

5. Learmount, D. "Flight Safety Foundation's European Aviation Safety Seminar," *Flight International* (March 20-26, 2001) 17.

6. Leveson, Nancy G. *Safeware: System Safety and Computers*. Boston: Addison Wesley, 1985.

7. Leveson, Nancy G. "Intent Specifications: An Approach to Building Human-Centered Specifications," *IEEE Transactions on Software Engineering*, SE-26, no. 1 (January 2000) 15-35.

8. Leveson, Nancy G. "Evaluating Accidents Models using Recent Aerospace Accidents: Part I. Event-Based Models," MIT Technical Report 2001. http://sunnyday.mit.edu/accidents.

9. Leveson, Alfaro, Alvarado, Brown, Hunt, Jaffe, Joslyn, Pinnel, Reese, Samarziya, Sandys, Shaw, Zabinsky, "Demonstration of a Safety Analysis on a Complex System," paper presented at Software Engineering Laboratory Workshop, NASA Goddard, Maryland, USA, December 1997. (Full report can be found at fttp://sunnyday.mit.edu/papers.html).

10. Leveson, Nancy G., Mats Heimdahl, Holly Hildreth, and Jon Damon Reese. "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, SE-20, no. 9 (September 1994) 684-707.

11. Leveson, Villepin, Daouk, Bellingham, Srinivasan, Neogi, Bachelder, Flynn, and Pilon, "A Safety and Human-Centered Approach to Developing New Air Traffic Management Tools." To appear in *Proceedings of ATM 2001 Conference*, New Mexico, December 2001.

12. Mackall, Dale A. National Aeronautics and Space Administration. November 1988. *Development and Flight Test Experiences with a Flight-Critical Digital Control System*. NASA Technical Paper 2857. Dryden Flight Research Facility, California, USA.

13. Miller, C.O. "A Comparison of Military and Civil Approaches to Aviation System Safety," *Hazard Prevention*, (May/June 1985) 29--34.

14. Sarter, N.D., D.D. Woods, and C.E. Billings, "Automation Surprises," in *Handbook of Human Factors/Ergonomics*, 2nd Edition, ed. G. Salvendy (New York: John Wiley & Sons, 1997)

Biography

Nancy G. Leveson, Ph.D., Professor, MIT, Aeronautics & Astronautics Dept., 33-315, 77 Massachusetts Ave., Cambridge MA 02139, USA, telephone - (617) 258-0505, facs. (617) 253-7397, e-mail - leveson@sunnyday.mit.edu.

Paper Release Form
19th International System Safety Conference

Title of Paper: _____

_____

I hereby authorize the System Safety Society to publish the paper listed above in the Proceedings of the 18th International System Safety Conference. Further, I agree to the following policy and notice regarding copyrights.

It is the policy of the System Safety Society, the sponsor of the International System Safety Conference, not to copyright the proceedings in order to provide the widest access for academic and educational use. Authors are free to copyright their papers as long as they agree with this policy. The policy to be contained in the proceedings is as follows:

Permission to print or copy: The copyright of all materials and commentaries published in these proceedings rests with the authors. Reprinting or copying for academic or educational use is encouraged and no fees are required; however, such permission is contingent upon giving full and appropriate credit to the author and the source of publication.

Author:
_____

Address:
_____
_____

Work Phone: _____
Home Phone: _____
FAX: _____
E-Mail: _____

_____
Signature Date

>>>>>>>>>>

Author:
_____

Address:
_____
_____

Work Phone: _____
Home Phone: _____
FAX: _____
E-Mail: _____

_____
Signature Date

Author:
_____

Address:
_____
_____

Work Phone: _____
Home Phone: _____
FAX: _____
E-Mail: _____

_____
Signature Date

>>>>>>>>>>

Author:
_____

Address:
_____
_____

Work Phone: _____
Home Phone: _____
FAX: _____
E-Mail: _____

_____
Signature Date

Mail to: John Livingston
Boeing Reusable Space Systems
555 Discovery Drive
Mail Code ZA-12
Huntsville, AL 35806-2809
(256) 971-3005, fax (256) 971-2699
john.m.livingston@boeing.com