The Role of Software in Spacecraft Accidents*

Nancy G. Leveson†
Aeronautics and Astronautics Department
Massachusetts Institute of Technology

1 Introduction

Software is playing an increasingly important role in aerospace systems. Is it also playing an increasing role in accidents and, if so, what type of role? In the process of a research project to evaluate accident models, I looked in detail at a variety of aerospace accidents that in some way involved software [1, 2] and was surprised at the similarity of the factors contributing to these losses.[1] To prevent accidents in the future, we need to attack these problems.

The spacecraft 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; and the loss of contact with the SOHO spacecraft in 1998.

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. These weaknesses not only contributed to the accident being investigated—for the Space Shuttle Challenger that might be a flawed design of the O-rings—but also can affect future accidents. For Challenger, the latter includes flawed decision making, poor problem reporting, lack of trend analysis, a "silent" or ineffective safety program, communication problems, etc.

The accidents are first briefly described for those unfamiliar with them, and then the common factors are identified and discussed. These factors are divided into three groups: (1) flaws in the safety culture, (2) management and organizational problems, and (3) technical deficiencies.

2 The Accidents

Ariane 501

On June 4, 1996, the maiden flight of the Ariane 5 launcher ended in failure. About 40 seconds after initiation of the flight sequence, at an altitude of 2700 m, the launcher veered off its flight path, broke up, and exploded. The accident report describes what they called the "primary cause" as the complete loss of guidance and attitude information 37 seconds after start of the

*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.

†Mailing Address: Professor Nancy Leveson, Room 33-313, Dept. of Aeronautics and Astronautics, MIT, 77 Massachusetts Ave., Cambridge MA 02139; Tel: 617-258-0505, email: leveson@mit.edu.

[1]The detailed analyses of these accidents and their causes using a hierarchical accident model can be found in [1, 2].

main engine ignition sequence (30 seconds after liftoff) [3]. The loss of information was due to specification and design errors in the software of the inertial reference system. The software was reused from the Ariane 4 and included functions that were not needed for Ariane 5 but were left in for "commonality." In fact, these functions were useful but not required for the Ariane 4 either.

Mars Climate Orbiter (MCO)

The Mars Climate Orbiter (MCO) was launched December 11, 1998 atop a Delta II launch vehicle. Nine and a half months after launch, in September 1999, the spacecraft was to fire its main engine to achieve an elliptical orbit around Mars. It then was to skim through the Mars upper atmosphere for several weeks, in a technique called aerobraking, to move into a low circular orbit. On September 23, 1999, the MCO was lost when it entered the Martian atmosphere in a lower than expected trajectory. The investigation board identified what it called the "root" cause of the accident as the failure to use metric units in the coding of a ground software file used in the trajectory models [4]. Thruster performance data was instead in English units.

Mars Polar Lander (MPL)

Like MCO, Mars Polar Lander (MPL) was part of the Mars Surveyor program. It was launched January 3, 1999, using the same type of Delta II launch vehicle as MCO. Although the cause of the MPL loss is unknown, the most likely scenario is that the problem occurred during the entry, deployment, and landing (EDL) sequence when the three landing legs were to be deployed from their stowed condition to the landed position [5, 6]. Each leg was fitted with a Hall Effect magnetic sensor that generates a voltage when its leg contacts the surface of Mars. The descent engines were to be shut down by a command initiated by the flight software when touchdown was detected. The engine thrust must be terminated within 50 milliseconds after touchdown to avoid overturning the lander. The flight software was also required to protect against a premature touchdown signal or a failed sensor in any of the landing legs.

The touchdown sensors characteristically generate a false momentary signal at leg deployment. This behavior was understood and the flight software should have ignored it. The software requirements did not specifically describe these events, however, and consequently the software designers did not account for them. It is believed that the software interpreted the spurious signals generated at leg deployment as valid touchdown events. When the sensor data was enabled at an altitude of 40 meters, the software shut down the engines and the lander free fell to the surface, impacting at a velocity of 22 meters per second (50 miles an hour) and was destroyed.

Titan/Centaur/Milstar

On April 30, 1999, a Titan IV B-32/Centaur TC-14/Milstar-3 was launched from Cape Canaveral. The mission was to place the Milstar satellite in geosynchronous orbit. An incorrect roll rate filter constant zeroed the roll rate data, resulting in the loss of roll axis control and then yaw and pitch control. The loss of attitude control caused excessive firings of the reaction control system and subsequent hydrazine depletion. This erratic vehicle flight during the Centaur main engine burns in turn led to an orbit apogee and perigee much lower than desired, placing the Milstar satellite in an incorrect and unusable low elliptical final orbit instead of the intended geosynchronous orbit.

The accident investigation board concluded that failure of the Titan IV B-32 mission was due to a failed software development, testing, and quality assurance process for the Centaur upper stage [7]. That failed process did not detect the incorrect entry by a flight software engineer of a roll rate filter constant into the Inertial Navigation Unit software file.

The roll rate filter itself was included early in the design phase of the first Milstar spacecraft, but the spacecraft manufacturer later determined that filtering was not required at that frequency. A decision was made to leave the filter in place for the first and later Milstar flights for "consistency."

SOHO (SOlar Heliospheric Observatory)

SOHO was a joint effort between NASA and ESA to perform helioseismology and to monitor the solar atmosphere, corona, and wind. The spacecraft completed a successful two-year primary mission in May 1998 and then entered into its extended mission phase. After roughly two months of nominal activity, contact with SOHO was lost June 25, 1998. The loss was preceded by a routine calibration of the spacecraft's three roll gyroscopes and by a momentum management maneuver.

The flight operations team had modified the ground operations procedures as part of a ground systems reengineering effort to reduce operations costs and streamline operations, to minimize science downtime, and to conserve gyro life. Though some of the modifications were made at the request of the SOHO science team, they were not necessarily driven by any specific requirements changes. A series of errors in making the software changes along with errors in performing the calibration and momentum management maneuver and in recovering from the emergency safing mode led to the loss of telemetry [8]. Communication with the spacecraft was never restored.

3 Flaws in the Safety Culture

The *safety culture* is the general attitude and approach to safety reflected by those working in an industry. Safety culture flaws reflected in the accident reports include complacency and discounting the risks associated with software, confusing safety with reliability in software-intensive systems, assuming risk decreases over time, and ignoring warning signs.

3.1 Complacency and Discounting of Risks

Success is ironically one of the progenitors of accidents when it leads to overconfidence and cutting corners or making tradeoffs that increase risk. This phenomenon is not new, and it is extremely difficult to counter when it enters the engineering culture in an organization. Complacency is the root cause of most of the other accident factors described in this paper and was exhibited in all the accidents studied.

The Mars Climate Orbiter (MCO) report noted that because JPL's navigation of interplanetary spacecraft had worked well for 30 years, there was widespread perception that "orbiting Mars is routine" and inadequate attention was devoted to risk management. A similar culture apparently permeated the Mars Polar Lander (MPL) project.

In the SOHO loss, overconfidence and complacency, according to the accident report, led to inadequate testing and review of changes to ground-issued software commands to the spacecraft, a false sense of confidence in the team's ability to recover from a safe-hold mode (emergency sun reacquisition) from which a recovery sequence must be commanded and executed under ground operator control, the use of tight schedules and compressed timelines that eliminated any time to handle potential emergencies, inadequate contingency planning, responses to emergencies without taking the designed-in time to consider the options, etc. Protections built into the process, such as the review of critical decisions, were bypassed. After two previous SOHO spacecraft retreats to safe mode, the software and procedures were not reviewed because higher priority had been assigned to other tasks. The report concludes that the success in recovering from the previous safe mode entries led to overconfidence by the operations team in their ability to recover and a lack of appreciation of the risks involved in entering and recovering from the safing mode.

All the accidents involved systems built within an engineering culture that had unrealistic expectations about software and the use of computers. It is common for engineers to underestimate the complexity of most software and to overestimate the effectiveness of testing. The 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.

In the Titan/Centaur accident, there apparently was no checking of the correctness of the software after the standard testing performed during development. For example, on the day of the launch, the attitude rates for the vehicle on the launch pad were not properly sensing the earth's rotation rate (the software was consistently reporting a zero roll rate) but no one had the responsibility to specifically monitor that rate data or to perform a check to see if the software attitude filters were operating correctly. In fact, there were no formal processes to check the validity of the filter constants or to monitor attitude rates once the flight tape was actually loaded into the Inertial Navigation Unit at the launch site. Potential hardware failures are usually checked up to launch time, but it may have been assumed that testing removed all software errors and no further checks were needed.

Complacency can also manifest itself in a general tendency of management and decision makers to discount unwanted evidence of risk. A *culture of denial* [9] arises in which any evidence of significant risk is dismissed.

A recommendation common to several of the spacecraft reports was to pay greater attention to risk identification and management. The investigators found that the project management teams appeared primarily focused on meeting mission cost and schedule objectives and did not adequately focus on mission risk. As an example, a report on the MPL loss concludes that the pressure of meeting the cost and schedule goals resulted in an environment of increasing risk in which too many corners were cut in applying proven engineering practices and in the checks and balances necessary for mission success.

While management may express their concern for safety and mission risks, true priorities are shown during resource allocation. MCO and MPL were developed under very tight "Faster, Better, Cheaper" budgets. The Titan program office had cut support for monitoring the software development and test process by 50% since 1994 and had greatly cut the number of engineers working launch operations. Although budget decisions are always difficult when resources are reduced—and budgets are almost always less than is optimal—the first things to be cut are often system safety, system engineering, quality assurance, and operations, which are assigned a low priority and assumed to be the least critical parts of the project.

3.2 Confusing Safety and Reliability in Software-Intensive Systems

Throughout the accident reports, there is an emphasis on failures as the cause of accidents. But accidents involving software are much more likely, in the author's experience, to be *system accidents* that result from dysfunctional interactions among components, not from individual component failure. Almost all the software accidents known to the author have resulted from the computer doing something wrong rather than the computer hardware or software failing to operate at all. In fact, each of the software or hardware components may have operated according to its specification (i.e., they did not fail), but the combined behavior of the components led to disastrous system behavior. All the accidents investigated for this paper displayed some aspects of system accidents.

System accidents are caused by interactive complexity and tight coupling [10]. 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 (often controlled by software) cannot all 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. Note that the use of redundancy only makes the problem worse—the added complexity introduced by redundancy has resulted in accidents that otherwise might not have occurred.

Engineering activities must be augmented to reflect the ways that software contributes to accidents. Almost all software-related aerospace accidents can be traced back to flaws in the requirements specification and not to coding errors—the software performed exactly as the designers intended (it did not "fail"), but the designed behavior was not safe from a system viewpoint. There is not only anecdotal but some hard data to support this hypothesis. Lutz examined 387 software errors uncovered during integration and system testing of the Voyager and Galileo spacecraft [11]. She concluded that the software errors identified as potentially hazardous to the system tended to be produced by different error mechanisms than non-safety-related software errors. She showed that for these two spacecraft, the safety-related software errors arose most commonly from (1) discrepancies between the documented requirements specifications and the requirements needed for correct functioning of the system and (2) misunderstandings about the software's interface with the rest of the system.

Software and digital systems require changes to some important aspects of engineering practice. Not only are failures not random (if the term "failure" makes any sense when applied to something like software that is pure design separated from the physical realization of that design), but the complexity of most software precludes examining all the ways it could "misbehave." And the failure modes (the way it misbehaves) can be very different than for physical devices. The JPL Mars Polar Lander accident report, like others, recommends using FMECA (Failure Modes and Effects Analysis) and FTA (Fault Tree Analysis) along with appropriate redundancy to eliminate failures. But these techniques were developed to cope with random wearout failures in hardware and are not very effective against design errors, the only type of error found in software.[2] The Ariane 5 accident 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 in the Ariane 5's first flight when both the primary and backup (redundant) 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 experi-

[2]Although computer hardware can fail, software itself is pure design and thus all errors are design errors, even typos, which could be categorized as a type of inadvertent design error. One could argue that typos have a random aspect, but typos are usually caught in testing and have not contributed to a large number of spacecraft accidents. In any event, FTA and FMEA would not be very helpful in identifying potential software typos.

ments (see, for example, [12]). 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 [13].

The first step in handling system accidents is for engineers to recognize the need for change and to understand that system safety and component or even functional reliability are different qualities. For software especially, one does not imply the other. 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. In some cases, increasing component or system reliability actually decreases safety and vice versa.

3.3 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. In several of the accidents, quality and mission assurance as well as system engineering were also reduced or eliminated during operations because it was felt they were no longer needed or the resources were needed more elsewhere. In MCO, for example, the operations group did not even have a mission assurance manager.

During SOHO operations, there was a lack of analysis of prior emergency sun reacquisitions, inadequate staffing, no apparent mission assurance and quality assurance functions, inadequate attention paid to changes, etc. The SOHO Mission Management Plan required that the NASA Project Operations Director be responsible for programmatic matters, provide overall technical direction to the flight operations team, and interface with the ESA technical support director. The position had been descoped over time by NASA from a dedicated individual during launch and commissioning to one NASA individual spending less than 10% of his time tracking SOHO operations. ESA was to retain ownership of the spacecraft and to be responsible for its technical integrity and safety, but they were understaffed to perform this function in other than routine situations. It is very common to assume that risk is decreasing after an extended period of success and to let down one's guard.

In fact, risk usually increases over time, particularly in software-intensive systems, 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.

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 [14]. As operators became more familiar with the Therac-25 operation, they started to type faster, which triggered a software error that had not surfaced previously. Similar changes in pilot behavior have been observed as they become more familiar with automation.

Software also tends to be frequently changed and "evolves" over time, The more changes that are made to software, the more the original design erodes and the more difficult it becomes to make changes without introducing errors. In addition, the assumptions and rationale behind the design decisions are commonly not documented and are easily violated when the software is changed. Changes to software appear to be easy and complacency can rear its ugly head again.

While it is indeed easy to change software, it is very difficult to change it correctly. Modifications to the SOHO command procedures were subjected to very little testing and review, perhaps because they were considered to be minor. The Mars Climate Orbiter software was changed to include a new thruster equation but a 4.45 correction factor (the difference between the metric and imperial units), buried in the original code, was not noticed when the new vendor-supplied equation was used to update the software [16].

To prevent accidents, all changes to software must be thoroughly tested and, in addition, 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, are also 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 (an airborne collision avoidance system required on most commercial aircraft flying in U.S. airspace) 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. It also assumes that a system hazard analysis was performed during the original system development.

3.4 Ignoring Warning Signs

Warning signs almost always occur before major accidents. In several of the accidents considered here, warning signs existed that the software was flawed but they went unheeded.

Engineers noticed the problems with the Titan/Centaur software after it was delivered to the launch site, but nobody seemed to take them seriously. A deficiency report on the difficulty in using the SOHO telemetry data interface had been submitted four years prior to the spacecraft loss, but never resolved. The problems experienced with the Mars Climate Orbiter (MCO) software during the early stages of the flight did not seem to raise any red flags. During the first four months of the MCO mission, the ground software angular momentum desaturation (AMD) files were not used in the orbit determination process because of multiple file format errors and incorrect spacecraft attitude data specifications. Four months were required to fix the files. Almost immediately (within a week) it became apparent that the files contained anomalous data that was indicating underestimation of the trajectory perturbations due to desaturation events. Despite all these hints that there were serious problems in the software and perhaps the development process, reliance was still placed on the supposedly fixed software without manual checks or alternative calculations.

A good argument can be made that it is unfair to judge too harshly those who have ignored warnings: Too many accident warnings that prove to be unfounded may desensitize those in decision-making positions and result in real warnings being ignored. Hindsight is always 20/20, and warning signs are always easier to identify after the accident has occurred. Nevertheless, people have a tendency to disregard events that do not lead to major accidents. Indications of potential problems almost always occur before major losses, and accidents could be prevented if these warning signs could be identified and heeded.

4 Management and Organizational Factors

The five accidents studied during this exercise, as well as most other major accidents, exhibited common organizational and managerial flaws, notably. a diffusion of responsibility and authority, limited communication channels, and poor information flow.

4.1 Diffusion of Responsibility and Authority

In most of the accidents, there appeared to be serious organizational and communication problems among the geographically dispersed partners. Responsibility was diffused without complete coverage and without complete understanding by anyone about what all the groups were doing. Roles were not clearly allocated. Both the Titan and Mars '98 programs were transitioning to process "insight" from process "oversight." Just as the MPL reports noted that "Faster, Better, Cheaper" was not defined adequately to ensure that it meant more than simply cutting budgets, this change in management role from oversight to insight seems to have been implemented simply as a reduction in personnel and budgets without assuring that anyone was responsible for specific critical tasks. One of the results of faster-better-cheaper was a reduction in workforce while maintaining an expectation for the same amount of work to be accomplished. In many of these accidents, the people were simply overworked—sometimes driven by their own dedication.

The MCO report concludes that project leadership did not instill the necessary sense of authority and accountability in workers that would have spurred them to broadcast problems they detected so that those problems might be "articulated, interpreted, and elevated to the highest appropriate level, until resolved." The Titan/Centaur accident also shows some of these same symptoms.

For SOHO, a transfer of management authority to the SOHO Project Scientist resident at Goddard Space Flight Center left no manager, either from NASA or ESA, as the clear champion of spacecraft health and safety. Instead, the transfer encouraged management decisions that maximized science return over spacecraft risk. In addition, the decision structure for real-time divergence from agreed-upon ground and spacecraft procedures was far from clear. The flight operations staff was apparently able to change procedures without proper review.

Inadequate transition from development to operations played a role in several of the accidents. Engineering management sometimes has a tendency to focus on development and to put less effort into planning the operational phase. The operations teams (in those accidents that involved operations) also seemed isolated from the developers. The MCO report notes this isolation and provides as an example that the operators did not know until long after launch that the spacecraft sent down tracking data that could have been compared with the ground data, which might have identified the software error while it could have been fixed. The operations crew for the Titan/Centaur also did not detect the obvious software problems, partly because of a lack of the knowledge required to detect them.

Most important, responsibility for safety does not seem to have been clearly defined outside of the quality assurance function on any of these programs. All the accident reports (except the Titan/Centaur) are surprisingly silent about their safety programs. One would think that the safety activities and why they had been ineffective would figure prominently in the reports.

Safety was originally identified as a separate responsibility by the Air Force during the ballistic missile programs of the 50s and 60s to solve exactly the problems seen in these accidents—to make sure that safety is given due consideration in decisions involving conflicting pressures and that safety issues are visible at all levels of decision making. An extensive system safety program was developed by NASA after the Apollo launch pad fire in 1967. But the Challenger accident report noted that the system safety program had become "silent" over time and through budget cuts. Has this perhaps happened again? Or are the system safety efforts just not handling software effectively?

One common mistake is to locate the safety efforts within the quality assurance function. Placing safety *only* under the assurance umbrella instead of treating it as a central engineering concern is not going to be effective, as has been continually demonstrated by these and other accidents. While safety is certainly one property (among many) that needs to be assured, it cannot be engineered into a design through after-the-fact assurance activities alone.

Having an effective safety program cannot prevent errors of judgment in balancing conflicting safety, schedule, and budget constraints, but it can at least make sure that decisions are informed and that safety is given due consideration. It also ensures that someone is focusing attention on what the system is not supposed to do, i.e., the hazards, and not just on what it is supposed to do. Both perspectives are necessary if safety is to be optimized.

4.2 Limited Communication Channels and Poor Information Flow

In the Titan/Centaur and Mars Climate Orbiter accidents, there was evidence that a problem existed before the loss occurred, but there was no communication channel established for getting the information to those who could understand it and to those making decisions or, alternatively, the problem-reporting channel was ineffective in some way or was simply unused.

All the accidents involved one engineering group not getting the information they needed from another engineering group. The MCO report cited deficiencies in communication between the project development team and the operations team. The MPL report noted inadequate peer communication and a breakdown in intergroup communication. The Titan/Centaur accident also involved critical information not getting to the right people. For example, tests right before launch detected the zero roll rate but there was no communication channel established for getting that information to those who could understand it. SOHO had similar communication problems between the operations team and technical experts. For example, when a significant change to procedures was implemented, an internal process was used and nobody outside the flight operations team was notified.

In addition, system engineering on several of the projects did not keep abreast of test results from all areas and communicate the findings to other areas of the development project: Communication is one of the most important functions in any large, geographically distributed engineering project and must be carefully planned and fostered.

Researchers have found that the second most important factor in the success of any safety program (after top management concern) is the quality of the hazard information system. Both collection of critical information as well as dissemination to the appropriate people for action is required. The MCO report concludes that lack of discipline in reporting problems and insufficient followup was at the heart of the mission's navigation mishap. In the Titan/Centaur loss, the use of voice mail and email implies there either was no formal anomaly reporting and tracking system or the formal reporting procedure was not known or used by the process participants for some reason. The report states that there was confusion and uncertainty as to how the roll rate anomalies should be reported, analyzed, documented and tracked because it was a “concern” and not a “deviation.” There is no explanation of these terms.

In all the 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 email. Large projects have successfully implemented informal email processes for reporting anomalies and safety concerns or issues. 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.

5 Technical Deficiencies

These cultural and managerial flaws manifested themselves in the form of technical deficiencies: (1) inadequate system and software engineering, (2) inadequate review activities, (3) ineffective system safety engineering, (4) inadequate cognitive engineering, and (5) flaws in the test and simulation environments.

5.1 Inadequate System and Software Engineering

For any project as complex as those involved in these accidents, good system engineering is essential for success. In some of the accidents, system engineering resources were insufficient to meet the needs of the project. In others, the process followed was flawed, such as in the flowdown of system requirements to software requirements or in the coordination and communication among project partners and teams.

In the Titan project, there appeared to be nobody in charge of the entire process, i.e., nobody responsible for understanding, designing, documenting, controlling configuration, and ensuring proper execution of the process. The Centaur software process was developed early in the Titan program and many of the individuals who designed the original process were no longer involved in it due to corporate mergers and restructuring and the maturation and completion of the Titan/Centaur design and development. Much of the system and process history was lost with their departure and therefore nobody knew enough about the overall process to detect that it omitted any testing with the actual load tape or knew that the test facilities had the capability of running the type of test that could have caught the error.

Preventing *system* accidents falls into the province of *system* engineering—those building individual components have little control over events arising from dysfunctional interactions among components. As the systems we build become more complex (much of that complexity being made possible by the use of computers), system engineering will play an increasingly important role in the engineering effort. In turn, system engineering will need new modeling and analysis tools that can handle the complexity inherent in the systems we are building. Appropriate modeling methodologies will have to include software, hardware and human components of systems.

Given that software played a role in all the accidents, it is surprising the reports reflected so little investigation of the practices that led to the introduction of the software flaws and a dearth of recommendations to fix them. In some cases, software processes were declared in the accident reports to have been adequate when the evidence shows they were not.

These accidents all involved very common system and software engineering problems, including poor specification practices, unnecessary complexity and software functions, software reuse without appropriate safety analysis, and violation of basic safety engineering design practices in the digital components.

Poor or Missing Specifications

The vast majority of software-related accidents have been related to flawed requirements and misunderstanding about what the software should do. This experiential evidence points to a need for better specification review and analysis—the system and software specifications must be reviewable and easily understood by a wide range of engineering specialists.

All the reports refer to inadequate specification practices. The Ariane accident report mentions poor specification practices in several places and notes that the structure of the documentation obscured the ability to review the critical design decisions and their underlying rationale. Inadequate documentation of design rationale to allow effective review of design decisions is a very common problem in system and software specifications [3]. The Ariane report recommends that justification documents be given the same attention as code and that techniques for keeping code and its justifications consistent be improved.

The MPL report notes that the system-level requirements document did not specifically state the failure modes the requirement was protecting against (in this case possible transients) and speculates that the software designers or one of the reviewers might have discovered the missing requirement if they had been aware of the rationale underlying the requirements. The small part of the requirements specification shown in the accident report (which may very well be misleading) seems to avoid all mention of what the software should not do. In fact, standards and industry practices often 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.

Not surprising, the interfaces were a source of problems. It seems likely from the evidence in several of the accidents that the interface documentation practices were flawed. The MPL report includes a recommendation that in the future "all hardware inputs to the software must be identified . . . The character of the inputs must be documented in a set of system-level requirements." This information is usually included in the standard interface specifications, and it is surprising that it was not.

There are differing accounts of what happened with respect to the MCO incorrect units problem. The official accident report seems to place blame on the programmers and recommends that the software development team be provided additional training in "the use and importance of following the Mission Operations Software Interface Specification (SIS)." Although it is not included in the official NASA Mars Climate Orbiter accident report, James Oberg in an IEEE Spectrum article on the accident [15] claims that JPL never specified the units to be used. It is common for specifications to be incomplete or not to be available until late in the development process. A different explanation for the error was provided by the developers [16]. According to them, the files were required to conform to a Mars Global Surveyor (MGS) heritage software interface specification. The equations used in the erroneous calculation were supplied by the vendor in English units.

> Although starting from MGS-heritage software, the coded MGS thruster equation had to be changed because of the different size RCS thruster that MCO employed (same vendor). As luck would have it, the 4.45 conversion factor, although correctly included in the MGS equation by the previous development team, was not immediately identifiable by inspection (being buried in the equation) or commented in the code in an obvious way that the MCO team recognized it. Thus, although the SIS required SI units, the new thruster equation was inserted in the place of the MGS equation—without the conversion factor[16].

This explanation raises questions about the other software specifications, including the requirements specification, which seemingly should include descriptions of the computations to be used. Either these did not exist or the software engineers did not refer to them when making the change. Formal acceptance testing apparently did not use the software interface specification because the test oracle (computed manually) used for comparison contained the same error as the output file [16] and thus testing did not detect the error. Similarly, informal interface testing by the Navigation team of ground software changes after launch did not test the correctness of the file formats (consistency with the interface specification) but concentrated on making sure the file could be moved across on the file server [16].

Complete and understandable specifications are not only necessary for development, but they are critical for operations and the handoff between developers, maintainers, and operators. In the Titan/Centaur accident, nobody other than the control dynamics engineers who designed the roll rate constants understood their use or the impact of filtering the roll rate to zero. When discrepancies were discovered right before the Titan/Centaur/Milstar launch, nobody understood them. The MCO operations staff also clearly had inadequate understanding of the automation and therefore were unable to monitor its operation effectively. The SOHO accident report mentions that no hard copy of the software command procedure set existed and the latest versions were stored electronically without adequate notification when the procedures were modified. The report also states that the missing software *enable* command (which led to the loss) had not been included in the software module due to a lack of system knowledge of the person who modified the procedure: he did not know that an automatic software function must be re-enabled each time Gyro A was despun. Such information, particularly about safety-critical features, obviously needs to be clearly and prominently described in the system specifications.

In some cases, for example the problems with the Huygens probe from the Cassini spacecraft, the designs or parts of designs are labeled as "proprietary" and cannot be reviewed by those who could provide the most important input [17]. Adequate system engineering is not possible when the system engineers do not have access to the complete design of the spacecraft.

Good specifications that include requirements tracing and design rationale are critical for complex systems, particularly those that are software-controlled. And they must be reviewable and reviewed in depth by domain experts.

Unnecessary Complexity and Software Functionality

One of the most basic concepts in engineering critical systems is to "keep it simple."

> The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay [18].

The seemingly unlimited ability of software to implement desirable features often, as in the case of most of the accidents examined in this paper, usually pushes this basic principle into the background: *Creeping featurism* is a common problem in software-intensive systems.

The Ariane and Titan/Centaur 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 software alignment function in the reused Ariane 4 software had no use in the different Ariane 5 design. The alignment function was designed to cope with the unlikely event of a hold in the Ariane 4 countdown: the countdown could be restarted and a short launch window could still be used. The feature had been used once (in 1989 in flight 33 of the Ariane 4). The Ariane 5 has a different preparation sequence and cannot use the feature at all. In addition, the alignment function computes meaningful results only before liftoff—during flight, it serves no purpose but the problem occurred while the function was operating after takeoff in the Ariane 5. The Mars Polar Lander accident also involved software that was executing when it was not necessary to execute, although in that case the function was required at a later time in the descent sequence.

The Titan/Centaur accident 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 same justification is used to explain why the unnecessary software function leading to the loss of the Ariane 5 was retained from the Ariane 4 software. Neither report explains why consistency was assigned such high priority. While changing software that works can increase risk, executing unnecessary software functions is also risky.

For SOHO, there was no reason to introduce a new function into the module that eventually led to the loss. A software function already existed to perform the required maneuver and could have been used. There was also no need to despin Gyro A between gyro calibration and the momentum maneuvers. In all these projects, tradeoffs were obviously not considered adequately (considering the consequences), 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.

Software Reuse without Appropriate Safety Analysis

Reuse and the use of commercial off-the-shelf software (COTS) is common practice today in embedded software development. The Ariane 5 software involved in the loss was reused from the Ariane 4. According to the MCO developers [16], the small forces software was reused from the Mars Global Surveyor project, with the substitution of a new thruster equation. Technical management accepted the "just like MGS" argument and did not focus on the details of the software.

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 described earlier: 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, i.e., it reliably performs the wrong function.

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—another reason to avoid unnecessary functions.

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 specific project risk—ignoring the potential safety issues associated with COTS software can lead to accidents and potential losses that are greater than the additional cost would have been to design and build new components instead of buying them.

If software reuse and the use of COTS components are 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 nor the need for difficult tradeoff decisions, and it almost always involves higher costs despite the common myth that introducing automation, particularly digital automation, will save money.

Violation of Basic Safety Engineering Practices in the Digital Components

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 (as occurred in 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. Runtime reasonableness and other types of checks should be part of the design criteria used for any real-time software.

5.2 Inadequate Review Activities

General problems with the way quality and mission assurance are practiced were mentioned in several of the reports. QA often becomes an ineffective activity that is limited simply to checking boxes signifying the appropriate documents have been produced without verifying the quality of the contents. The Titan/Centaur accident report makes this point particularly strongly.

Review processes (outside of QA) are also described as flawed in the reports but few details are provided to understand the problems. The Ariane 5 report states that reviews including all major partners in the Ariane 5 program took place, but no information is provided about what types of reviews were held or why they were unsuccessful in detecting the problems. The MCO report recommends that NASA "conduct more rigorous, in-depth reviews of the contractor's and team's work," which it states were lacking on the MCO. The report also concludes that the operations team could have benefited from independent peer reviews to validate their navigation analysis technique and to provide independent oversight of the trajectory analysis. There is no mention of software quality assurance activities or the software review process in the MCO report.

In the MPL descent engine control software reviews, apparently nobody attending was familiar with the potential for spurious Hall Effect sensor signals. There have also been cases where concerns about proprietary software have prevented external reviewers familiar with the spacecraft from a systems viewpoint from reviewing the software (see, for example, the Cassini/Huygens communications link enquiry board report [17]).

The SOHO accident report states that the changes to the ground-generated commands were subjected to very limited review. The flight operations team placed high reliance on ESA and Matra Marconi Space representatives who were quite knowledgeable about the spacecraft design, but there were only two of them and neither was versed in the computer language used to define the commands. A simulation was performed on the new compressed SOHO timelines, but the analysis of a problem detected during simulation was still going on as the new procedures were being used.

The Ariane report 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. An assumption by the Ariane 5 developers that it was not possible to perform a complete system integration test made simulation and analysis even more important, including analysis of the assumptions underlying any simulation.

The Titan/Centaur report was the only one to mention the existence of an independent verification and validation review process by a group other than the developers. In that process, default values were used for the filter rate constants and the actual constants used in flight were never validated.

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.

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.

As mentioned earlier, almost all software-related accidents have involved incomplete requirements specification and unhandled or mishandled system states or conditions. 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. Such errors are most likely to be found if spacecraft and subsystem experts participate actively in the reviews.

Software hazard analysis and requirements analysis techniques and tools exist to assist in finding 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 can be subjected to a later software design review by a different set of reviewers). The only information relevant for a software requirements review is the software behavior that is visible outside the computer. Specifying only blackbox behavior (in engineering terminology, the *transfer function* across the digital component) allows the 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 also critical to the success of such a review. The best way to find errors in the software requirements is to include a wide range of disciplines and expertise in the review process. They must be able to read and understand the specifications without extensive training and, ideally, the notation should not differ significantly from standard engineering notations. While formal and executable specification languages have tremendous potential for enhancing our ability to understand the implications of complex software behavior and to provide correct and complete requirements, most of the languages created by computer scientists require too much reviewer training to be practical. A high priority on readability and learnability has not been placed on the development of such languages.

5.3 Ineffective System Safety Engineering

All of the accident reports studied are surprisingly silent about the safety programs and system safety activities involved. Take SOHO for example. A hazard analysis surely would have shown that the roll rate and the status of gyros A and B were critical, and this information could have guided the design of feedback channels to the operators about their status. A rigorous system safety process also would have triggered special safety analysis when changes were made to the SOHO operational procedures involving safety-critical components. In addition, a strong system safety program would have ensured that high priority was given to the analysis of previous emergency sun reacquisitions, that greater controls were placed on safety-critical operational procedures, and that safety-related open operational reports, such as the one reporting the difficulty the SOHO operators were having in reviewing telemetry data, did not stay open for four years and instead were tracked and resolved in a timely manner.

There did appear to be a criticality analysis performed on many of these projects, albeit a flawed one. Several of the reports recommend reconsidering the definition they used of critical components, particularly for software. Unfortunately, not enough information is given about how the criticality analyses were performed (or in some cases if they were done at all) to determine why they were unsuccessful. Common practice throughout engineering, however, is to apply the same techniques and approaches that were used for electromechanical systems (e.g., FMEA and FMECA) to the new software-intensive systems. This approach will be limited because the contribution of software to accidents, as noted previously, is different than that of purely mechanical or electronic components. In particular, software does not fail in the sense assumed by these techniques.

Often hazard analyses simply omit software, and when included it is often treated superficially at best. The hazard analysis produced after the Mars Polar Lander loss is typical. The JPL report on the loss identifies the hazards for each phase of the entry, descent, and landing sequence, such as *Propellant line ruptures*, *Excessive horizontal velocity causes lander to tip over at touchdown*, and *Premature shutdown of the descent engines.* For software, however, only one hazard—*Flight software fails to execute properly*—is identified, and it is labeled as common to all phases.

The problem with such vacuous statements about software hazards is that they provide no useful information—they are equivalent to simply substituting the single statement *Hardware* *fails to operate properly* for all the other identified system hazards. What can engineers do with such general statements? Singling out the JPL engineers here is unfair because the same types of useless statements about software are common in the fault trees and other hazard analyses found in almost all organizations and industries. The common inclusion of a box in a fault tree or failure analysis that says simply *Software Failure* or *Software Error* can be worse than useless because it is untrue—all software misbehavior will not cause a system hazard in most cases—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 such a figure does not reflect in any way the probability of the software exhibiting a particular hazardous behavior.

Software by itself is never dangerous—it is an abstraction without the ability to produce energy and thus to lead directly to a physical loss. Instead, it contributes to accidents through issuing (or not issuing) instructions to other components in the system. In the case of the identified probable factor in the MPL loss, the dangerous behavior was *Software prematurely shuts down the descent engines.* Such an identified unsafe behavior would be much more helpful during development in identifying ways to mitigate risk than the general statement *Software fails to execute properly.*

There are several instances of flawed risk tradeoff decisions associated with these accidents. For example, in the Ariane accident, there was a lack of effective analysis to determine which software variables should be protected during execution. Unfortunately, the accident reports describe flawed decisions, but not the process for arriving at them. Important information that is missing includes how the analyses and trade studies were performed and what additional information or additional analysis techniques could have allowed better decisions to be made.

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 [19]. 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 description of the MCO problem by the developers [16] says that the best chance to find and eliminate the problem existed at the early stages of development, but the team failed to recognize the importance of the small forces ground software and it was not given the same attention as the flight software.

The Titan/Centaur accident provides another 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 (a critical factor in the 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.

Not only is such a *fly-fix-fly* approach inadequate for complex systems in general, particularly when a single loss is unacceptable, but considering only the specific events and conditions occurring in past accidents is not going to be effective when new technology is introduced into a system. Computers are, in fact, introduced in order to make radical changes in functionality and design. In addition, software is often used precisely because it is possible to make changes for each mission and throughout operations—the system being flown today is often not the same one that existed yesterday. 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 the original development and when making changes during operations.

At the same time, system-safety techniques, like other engineering techniques, need to be expanded to include software and the complex cognitive decision making and new roles played by human operators [14]. Existing approaches need to be applied, and new and better ones developed. Where appropriately modified system safety techniques have been used, they have been successful. If system hazard analysis is performed prior to software implementation (not just prior to testing, as is recommended in the MPL report), requirements can be analyzed for hazardous states and protection against potentially hazardous behavior designed into the software logic from the beginning.

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.

One of the benefits of using system-safety engineering processes is simply that someone becomes responsible for ensuring that particular hazardous behaviors are eliminated if possible or their likelihood reduced and their effects mitigated in the design. Almost all attention during development is focused on what the system and software are supposed to do. A system safety engineer or software safety engineer is responsible for ensuring that adequate attention is also paid to what the system and software are *not* supposed to do and verifying that hazardous behavior will not occur. It is this unique focus that has made the difference in systems where safety engineering successfully identified problems that were not found by the other engineering processes.

5.4 Flaws in the Test and Simulation Environments

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. It is usually difficult, however, 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, particularly 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 and flaws 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. Testing of SOHO operational procedures was primarily performed using a simulator, but the simulator had not been maintained with all the on-board software changes that had been implemented on the spacecraft, essentially making such testing useless.

For both the Ariane 5 and Mars '98 projects, a conclusion was reached during development that the components implicated in the accidents could not be tested and simulation was substituted. After the fact, it was determined that such testing was indeed possible and would have had the ability to detect the design flaws. The same occurred with the Titan/Centaur accident, where default and simulated values were used in system testing although the real roll rate filter constants could have been used. Like Ariane, the Titan/Centaur engineers incorrectly thought the rigid-body simulation of the vehicle would not exercise the filters sufficiently. Even the tests performed on the Titan/Centaur right before launch (because anomalies had been detected) used default values and thus were unsuccessful in detecting the error. After wiring errors were discovered in the MPL testing process, for undisclosed reasons the tests necessary to detect the software flaw were not rerun.

Not all problems in testing can be traced to the simulation environment, of course. There have been cases of spacecraft losses involving inadequate, inappropriate, and ill-suited testing. A basic problem is one of piecemeal testing and not testing at the system level for system-level effects and emergent behavior. The rush to get the ground software operational after a problem was discovered post-launch in MCO resulted in the testing program being abbreviated. At a November 10, 1999 press conference, Alfred Stephenson, the chief accident investgator, admitted, "Had we done end-to-end testing, we believe this error would have been caught." But the rushed and inadequate preparations left no time to do it right. The problem lies not only in testing, but in relying on software that had not been adequately tested without additional manual or other checks to gain confidence. The same lack of system test also contributed to the WIRE (Wide-Field Infrared Explorer Mission) spacecraft where a contributing cause cited in the accident report was that no system level end-to-end test with live pyrotechnic devices in the as-flown configuration had been done [20].

A final very common problem in software testing is inadequate emphasis on off-nominal and stress testing.

Better system testing practices are needed for components containing software (almost everything these days), more accurate simulated environments need to be used in software testing, and the assumptions used in testing and simulations need to be carefully checked.

6 Inadequate Cognitive Engineering

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: mode confusion and situational awareness difficulties; inadequate feedback to support effective monitoring and decision making; 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. 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 [21, 22]. Not all of this information seems to have affected engineering practice: After commercial aircraft accidents, 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).

As more sophisticated automation has been introduced into spacecraft control and control of safety-critical functions is increasingly shared between humans and computers, the same problems found in high-tech aircraft are appearing. Neither the Mars Climate Orbiter 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 these problems.

Problems abound in the design of the interfaces between humans and automation. The SOHO operations personnel had filed a report (which had not been resolved in four years) about the difficulty they were having interpreting the telemetry data using the interface they were given. In addition, several places in the SOHO accident report hint at the controllers not having the information they needed about the state of the gyros and the spacecraft in general to make appropriate decisions. The misdiagnosis of Gyro B as the bad one and its subsequent deactivation raises many important questions about the information provided to the operators that are not answered in the accident report.

Complexity in the automation combined with poor documentation and training procedures are contributing to the problems we are seeing. Sometimes the incorrect assumption is made that introducing computers lessens the need for in-depth knowledge by operational personnel but the opposite is true. Some of the spacecraft accidents where operators were implicated involved a failure to transfer skill and knowledge from those who operated spacecraft in prior missions, and they were therefore unable to detect the software deficiencies in time to save the mission.

Either the design of the automation we are building needs to be improved from a cognitive engineering viewpoint or new training methods are needed for those who must deal with the clumsy automation and confusing, error-prone interfaces we are designing.

7 Conclusions

Complacency and misunderstanding software and its risks were at the root of all these accidents. Software presents tremendous potential for increasing our engineering capabilities. At the same time, it introduces new causal factors for accidents and requires changes in the techniques used to prevent the old ones. We need to apply the same good engineering practices to software development that we apply to other engineering technologies while also understanding the differences and making the appropriate changes to handle them. There is no magic in software—it requires hard work and is difficult to do well, but the result is worth the effort.

References

[1] Leveson, N.G. "Evaluating Accident Models using Recent Aerospace Accidents: Part I. Event-Based Models". MIT Technical Report, 2001.

[2] Weiss, K., Leveson, N.G., Lundqvist, K., Farid, N., and Stringfellow, M. "An Analysis of Causation in Aerospace Accidents". *Digital Aviation Systems Conference*, Daytona, October 2001.

[3] Lions, J.L. "Ariane 501 Failure: Report by the Inquiry Board". European Space Agency, 19 July, 1996.

[4] Stephenson, A. "Mars Climate Orbiter: Mishap Investigation Board Report". NASA, November 10, 1999.

[5] JPL Special Review Board. "Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions". Nasa Jet Propulsion Laboratory, 22 March 2000.

[6] Young, T. "Mars Program Independent Assessment Team Report". NASA, March 14, 2000.

[7] Pavlovich, J.G. "Formal Report of Investigation of the 30 April 1999 Titan IV B/Centaur TC-14/Milstar-3 (B-32) Space Launch Mishap". U.S. Air Force, 1999.

[8] NASA/ESA Investigation Board. "SOHO Mission Interruption". NASA, 31 August 1998

[9] Andrew Hopkins. *Managing Major Hazards: The lessons of the Moira Mine Disaster.* Allen & Unwin, 1999.

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

[11] Lutz, R.R. "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems". *Software Requirements Conference*, IEEE, January 1992.

[12] Knight, J.C. and Leveson, N.G. "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming". *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986, pp. 96-109.

[13] Mackall, D.A. "Development and Flight Test Experiences with a Flight-Critical Digital Control System". NASA Technical Paper 2857, National Aeronautics and Space Administration, Dryden Flight Research Facility, November 1988.

[14] Leveson, N.G. *Safeware: System Safety and Computers.* Addison Wesley, 1985.

[15] Oberg, J. "Why the Mars Probe Went Off Course". *IEEE Spectrum Magazine*, Vol. 36, No. 12, December 1999.

[16] Euler, E.E., Jolly, S.D., and Curtis, H.H. "The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved". *Guidance and Control 2001*, American Astronautical Society, paper AAS 01-074, 2001.

[17] D.C.R (Chairman). "Report of the Huygens Communications System Inquiry Board". NASA, December 2000.

[18] Hoare, A.A. "The Emperor's Old Clothes". *Communications of the ACM*, 24(2), pages 75-83, 1981.

[19] Frola, F.R. and Miller, C.O. "System Safety in Aircraft Acquisition". Logistics Management Institute, Washington D.C. January 1984.

[20] Branscome, D.R. (Chairman). "WIRE Mishap Investigation Board Report". NASA, June 8, 1999.

[21] Bureau of Air Safety Investigation. "Advanced Technology Aircraft Safety Survey Report". Department of Transport and Regional Development, Australia, June 1996.

[22] Sarter, N. D. and Woods, D. "How in the world did I ever get into that mode?": Mode error and awareness in supervisory control. *Human Factors 37*, 5–19.