AIAA 2001-4763

SYSTEMIC FACTORS IN SOFTWARE-RELATED SPACECRAFT ACCIDENTS*

Prof. Nancy G. Leveson
Aeronautics and Astronautics Dept.
Massachusetts Institute of Technology
leveson@mit.edu and http://sunnyday.mit.edu

ABSTRACT

Four relatively recent spacecraft accidents involving software are examined: 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; and the placing of a Milstar satellite in an incorrect and unusable orbit by a Titan IV B in 1999. Important common factors and similarities were identified.

INTRODUCTION

Some accident reports were examined in a research project to evaluate a new accident model. The model requires accounting for each event with related conditions that led to the event and then each condition with the systemic factors that explained the existence of the conditions. The complete report, which included aircraft accidents, can be found at http://sunnyday.mit.edu/accidents.

On the surface, the events involved in the four spacecraft accidents appear to be very different, except for the fact that software played a role in all of them. However, a more careful, detailed analysis of these accidents and the systemic factors involved showed striking similarities in the root causes.

The accidents are first briefly described for those unfamiliar with them and then the common factors identified and discussed.

ARIANE 5

On June 4, 1996, the maiden flight of the Ariane 5 launcher ended in failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of only 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 main engine ignition sequence (30 seconds after liftoff). 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 "consistency." In fact, these functions were useful but not required for the Ariane 4 either.

MARS CLIMATE ORBITER

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. Thruster performance data was in English units instead of the metric units specified in the software interface specification.

MARS POLAR LANDER

During the entry, deployment, and landing (EDL) sequence of the Mars Polar Lander (MPL), the three landing legs were to be deployed from their stowed condition to the landed position at an altitude of about 1500 meters. 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

*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.

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 properly account for them. According to the most likely accident scenario, 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 IV/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 excessing firings of the reaction control system and subsequent hydrazine depletion. Erratic vehicle flight during the Centaur main engine burns caused it to achieve an orbit apogee and perigee much lower than desired, which resulted in the Milstar being placed 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. 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.”

COMMON SYSTEMIC FACTORS

Eliminating the specific events and conditions involved in a particular accident may prevent a repetition of the same accident scenario, but not those with different detailed event scenarios stemming from the same root or systemic problems. Accidents rarely repeat themselves in exactly the same way and patches to particular parts of the system may be ineffective in preventing future accidents. For example, if another Space Shuttle accident occurs, it is unlikely to be caused by the exact same type of O-ring problem: An O-ring failure precipitated the loss, but the root causes identified in the accident investigation were related to an accumulation of problems, including an inadequate problem reporting system, inadequate trend analysis, misrepresentation of criticality, inadequate resources devoted to safety, lack of safety personnel involvement in important discussions and decisions, etc. Preventing accidents in the future requires fixing those systemic deficiencies and not just the symptoms that led to that specific accident scenario.

From the information provided in the accident reports for the accidents described, the following systemic factors can be identified. They are grouped into three categories: flaws in the safety culture, ineffective organizational structure and communication, and inadequate or ineffective technical activities.

I. Flaws in the Safety Culture

Overconfidence and complacency

Success is ironically one of the progenitors of accidents when it leads to overconfidence and cutting corners or making tradeoffs that increase risk. Petroski, in his book *To Engineer is Human*, describes this common phenomenon. It is not new, and it is extremely difficult to counter when it enters the engineering culture in an organization.

The Rogers Commission investigating the *Challenger* accident noted that Shuttle flights had become routine: safety margins were relaxed over time and risks were tolerated because they had been experienced before—no adequate attempt was made to eliminate them. NASA and its contractors accepted escalating risk (although they probably did not realize it was escalating) because they had gotten away with it previously. The Mars Climate Orbiter (MCO) report noted similarly 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.

Complacency can manifest itself in a general tendency of management and decision makers to discount unwanted evidence of risk. In an analysis of an accident in the Moura mine in Australia, Hopkins describes a general phenomenon he calls a *culture of denial* in which it is generally believed that there is no significant risk and production pressures lead to dismissing any evidence to the contrary.

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, true priorities are shown during resource allocation. By the time of the fatal Challenger flight, reductions had been made in the safety engineering functions that essentially made those functions ineffective. 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 desirable), 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.

Underestimating and Not Understanding Software Risks

Accidents involving software often occur within an engineering culture that has unrealistic expectations about software and the use of computers. The common misperceptions (or myths) take two forms: 1) Software does not fail and all errors will be eliminated through testing and (2) software can be handled using the same techniques as hardware.

The first myth stems from an underestimation of the complexity of most software and an overestimation of the effectiveness of testing. The Ariane 5 report notes that software was assumed to be correct until it was shown to be faulty. This form of complacency also plays a part in the common proliferation of software functionality and in unnecessary design complexity.

In the Titan 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 but no one had the responsibility to specifically monitor that rate data or to perform a check to see if the attitude filters were correctly sensing the earth's rotation rate. 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 INU 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. Even right before launch, the programmed rate check used a default set of constants to filter the measured rate rather then the actual constants loaded on the Centaur.

The second myth is that the same techniques used to make electromechanical systems safe or reliable will work in software-intensive systems. This myth leads to ineffective and inadequate technical activities directed toward safety. The recommended approach in the Mars Polar Lander report is an example. It and other reports recommend using FMECA or FTA along with appropriate redundancy to eliminate failures. But software and digital systems bring a totally different game to engineering practice. Some classically trained engineers have difficulty appreciating the new and very different engineering environment created by the introduction of software and the new mindset and approaches required. Not only are the failures not random (if the term "failure" makes any sense when applied to something that is pure design separate from the physical realization of that design), but the complexity of most software precludes examining all the ways it could "misbehave." In addition, the failure modes are very different than for physical devices.

Throughout the accident reports, there is an emphasis on failures as the cause of accidents. The contribution of software to accidents is very different than that of hardware, and engineering activities must be augmented to reflect this. Almost all software-related accidents can be traced back to flaws in the requirements specification and not to coding errors—the software performed exactly as specified, but the specification was incorrect.

Understanding these differences between software and other engineering "materials" and implementing alternative approaches that will be effective for software has been slow in engineering. For example, the JPL report on the MPL loss contains a figure that shows the "potential failure modes for the [entry, descent, and landing] sequence." Potential hardware failures or misbehaviors, such as *Propellant line rupture* or *Excessive horizontal-velocity causes lander to tip over at touchdown*, are identified for each stage except for software. Instead, a statement *Flight software fails to execute properly* is identified as common to all phases. The problem with this is that it provides no useful information—it is equivalent to simply substituting a single statement for all the other hazards identified in the figure with "hardware fails to execute properly." Singling out the JPL engineers is unfair here because the same types of useless statements about software can be found in almost all the fault trees and failure analyses I see in industry, and this practice is not limited to aerospace.

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 the risk than the general statement *Software fails to execute properly*.

Overrelying on Redundancy

Redundancy is commonly used to reduce component failures and increase system reliability. The *Challenger* accident is typical. The engineers and managers relied on redundancy without properly evaluating whether the redundancy provided adequate protection. Once the original evaluation had been completed, they continued to believe in the independence of failures of the redundant O-rings long after that independence assumption had been shown to be incorrect. While some of this mistaken reliance had a basis in inadequate communication and documentation procedures, reliance was placed on redundancy even by those who *knew* the criticality rating had been increased from 1R (highest criticality but risk mitigated by redundancy) to 1 (highest criticality).

Accidents are changing their character. This change is not solely the result of using digital components, but it is made possible because of the flexibility of software. Most of the accidents investigated in this research showed at least some aspects of *system* accidents, where all or most of the components implicated in the accident worked as specified but the combined behavior of the components led to disastrous system behavior. Not only did each component in isolation work correctly (i.e., they satisfied their specifications), but, in fact, many of the design features that contributed to the accident involved standard recommended practice. Protecting against software "failures" will do nothing to protect against system accidents.

Safety and reliability are *different* qualities for software—one does not imply the other. Although confusing reliability with safety is common in engineering, it is perhaps most unfortunate with regard to software as it encourages spending most of the effort devoted to safety on activities that are likely to have no effect.

In this regard, 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. 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 both the Inertial Reference System computers shut themselves down (exactly as they were designed to do) in response to 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. This independence assumption has been shown to be false in practice and in a large number of carefully designed experiments (see, for example, [1]). The multiple versions tend to produce incorrect results processing 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 [4].

Assuming Risk Decreases over Time

In the Milstar satellite loss, the Titan Program Office 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, the operations group did not have a mission assurance manager. The *Challenger* accident report also noted cuts in the operational safety activities (the "Silent Safety" program). It is very common to assume that risk is decreasing after repeated successes.

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 that 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. For example, 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. 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.

Thus we have the rather surprising conclusion that *as system error rates decrease and reliability increases, the risk of accidents may actually be increasing.*

Ignoring Warning Signs

Warning signs almost always occur before major accidents. For *Challenger*, warning signs existed but were ignored and concerned engineers were unable to draw attention to the O-ring problems. In several of the other spacecraft accidents, there were warning signs 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. The problems experienced with the MCO software during the early stages of the flight also did not seem to raise any red flags.

II. Ineffective Organizational Structure and Communication

The Ariane 5 report was strangely silent about organizational and communication issues. However, it includes a recommendation to "consider a more transparent organization of the cooperation among partners" and concludes that "close engineering cooperation, with clear cut authority and responsibility, is needed to achieve system coherence, with simple and clear interfaces between partners." No other information is provided so little can be learned about organizational factors from this accident. The other accident reports, however, all described classic management factors related to accidents.

Diffusion of Responsibility and Authority

In many 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 seems to have been implemented simply as a reduction in personnel and oversight responsibility without assurance that anyone was responsible for specific necessary tasks.

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 accident also shows some of these same symptoms.

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 of any project or system. 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 problems, partly because of a lack of the knowledge required to detect them. Better communications and involvement of the developers in the launch operations might have avoided the losses.

Most important, responsibility for safety does not seem to have been clearly defined outside of the quality assurance function on any of these programs. As noted in the Challenger accident report, 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. 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. 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.

Having an effective safety program cannot prevent errors in judgement in balancing conflicting requirements of risk, schedule, and cost, but it can at least make sure that decisions are informed and that risk is given due consideration.

Low-Level Status and Inappropriate Organizational Placement of the Safety Program

The reports on the recent accidents involving NASA projects 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. In fact, the only time system safety is mentioned is with respect to quality assurance. Is system safety engineering being performed at all on these projects? Perhaps it is only considered necessary for missions involving humans or safety is being confused with reliability. The MPL and MCO accident reports both lament the lack of "what-if" analysis, which is the hallmark of system safety engineering. It is very effective in preventing general losses, not just those involving human life.

More information is needed to determine why this type of analysis is not being used. System safety engineering techniques may not have been used on these projects or they may have been ineffective and system safety marginalized by being limited to the quality assurance program: 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.

Limited Communication Channels and Poor Information Flow

In the Titan, Challenger, 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 decision makers 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 accident also involved critical information not getting to the right people. For example, the 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.

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: Establishing and implementing this type of intergroup technical communication is one of the primary roles for system engineering.

Communication is one of the most important functions in any large, geographically distributed engineering project and must be carefully planned and fostered.

III. Ineffective or Inadequate Technical Activities

Although the actual technical errors made were different in each of the accidents, common flaws in the engineering activities led to these errors.

Inadequate System 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.

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. Such modeling and analysis techniques are currently only in their infancy.

Flawed Review Process

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 that the appropriate documents are produced without verifying the quality of the contents. The Titan 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. 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 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 will be (or 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 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 these projects.

None of the reports but Titan mentions any independent verification and validation (IV&V) review process (beyond normal system testing) by a group other than the developers. The Titan program did have an independent IV&V process, but it used only default values for the filter rate constants and never validated the actual constants used in flight.

Inadequate Specifications

Several of the reports refer to inadequate specification practices. The Ariane accident report 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 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.

In general, the vast majority of software-related accidents can be traced to flawed requirements rather than coding errors. Either (1) the requirements are incomplete or contain wrong assumptions about the operation of the controlled system or the required operation of the computer or (2) there are unhandled controlled-system states and environmental conditions. This experiential evidence leads directly 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 [3].

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 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 launch, nobody understood them. The MCO operations staff also clearly had inadequate understanding of the automation and therefore were unable to effectively monitor its operation. Good specifications that include requirements tracing and design rationale are critical for long-lived systems and may be able to improve the effectiveness of the operations personnel.

The Titan report stressed the lack of a different type of specification: formal documentation of the overall process flow. The Centaur software process was developed early in the Titan/Centaur program and many of the individuals who designed the original process are 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 LMA test facilities had the capability of running the type of test that could have caught the error.

Violation of Basic Safety Engineering Practices in the Digital Parts of the System Design

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 critical functions, eliminating unnecessary functionality, designing error-reporting messages such that they cannot be confused with critical data (see the Ariane accident), and reasonableness checking of inputs and internal states. Consider the MPL loss: 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. In the case of MPL, code was executing when it was not needed. The same was true for Ariane and for Titan. Each of these decisions was surely 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.

Inadequate Software Engineering

As a person trained in software engineering, I found the reports very frustrating in their lack of investigation of the practices that led to the introduction of the software flaws and the related lack of recommendations to fix them. In some cases, software processes were declared to be adequate when all evidence pointed to the fact that they were not. Only the Titan investigation board and to a lesser extent the Ariane 5 investigators looked at the software in enough detail to determine the conditions and systemic factors related to the software engineering process (or at least included this information in the accident report).

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. In the case of MCO, either the MCO programmers were incompetent (unlikely) or there is a more likely explanation for the units error such as the common practice of writing interface and other specifications after coding has begun, poorly designed specifications that make retrieving information error prone, software programmers without the necessary science background to understand the importance of the units used, inadequate information provided to the programmers for them to understand the role of the data in the larger system, etc.

There also appear to be problems in software reviews and perhaps also in implementing the special practices unique to real-time, embedded software. Why were unhandled cases not detected? Testing only against the requirements specification will not uncover errors in the specification. What type of requirements validation was performed? Why were defensive programming and exception-handling practices not used more effectively?

Without further investigation, we cannot learn enough about the software engineering practices involved in these accidents to prevent future reoccurrences. That is very unfortunate.

Flawed or Inadequate Analysis of Software Functions

Limitations in how thoroughly software can be tested make simulation and other types of analysis critical for software-intensive systems. The two identified MPL software errors involved incomplete handling of software states and are both examples of very common specification flaws and logic omissions. As such, they were potentially detectable by formal and informal analysis and review techniques. Software hazard analysis and requirements analysis techniques exist (and more should be developed) to detect this type of incompleteness.

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. The assumption by the Ariane 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. Executable and easy to read formal requirements specifications should theoretically be able to help here. Unfortunately, most of the formal specification languages that have been proposed use obscure and obtuse notations that are inappropriate for industry projects.

Software Reuse without Appropriate Analysis of its Safety

Reuse and the use of commercial off-the-shelf software (COTS) is common practice today in embedded software development. 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 a confusion between software reliability and safety. 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 using different environmental assumptions. *Changing the environment in which the software operates makes all previous usage experience with the software irrelevant for determining safety.* The same software flaw that led to the overdosing of five patients by the Therac-25 existed in the Therac-20 software (which was reused on the Therac-25), but the flaw had no untoward effects in the different Therac-20 system design.

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. For complex designs, the safety analysis required stretches the limits of current technology. For this type of analysis to be technically and financially feasible, reused software must contain only the features necessary to perform critical functions. As noted earlier, 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. Note, however, that COTS software is often constructed with as many features as possible to make it commercially useful in a variety of systems. Thus there is a tension between using COTS and being able to perform a safety analysis or 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 reuse and the use of COTS software 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, and it almost always involves higher costs, despite the common myth that introducing automation saves money.

Inadequate System Safety Engineering

Judging only from the information (or lack of information) included in the accident reports, none of these projects appear to have had adequate system safety engineering.

For example, 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 even *if* they were done) 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.

There appear to be several instances of flawed risk tradeoff decisions. For example, in the Ariane accident, there was a lack of effective analysis to determine which 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 [2]. When hazard analyses are not done, are done only after the fact (for example, as a part of quality or mission assurance), or are done but the information is never integrated into the engineering decision-making 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 hazard analysis is not done. The risk analysis, in that case, was not based on determining the steps critical to mission success (a traditional hazard analysis) but instead considered only the problems that had occurred in previous launches. Software constant generation was considered to be low risk because there had been no previous problems with it. Not only is such an approach inadequate for complex systems in general, 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 decision making.

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 [2]. Existing approaches need to be applied, and new and better ones developed. When 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 logic.

The MCO report and the Challenger report were the only ones to recommend instituting 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 these 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 reports, when changes were suggested, instead described classic reliability engineering approaches that are unlikely to be effective for system accidents or software-intensive systems.

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.

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. The Ariane and Titan accidents clearly involved software that was not needed, but surprisingly the decision to put in or to keep these features (in the cases of reuse) was not questioned in the accident reports. The MPL 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 filter was not needed but was kept in for "consistency." The exact same words are used for the Ariane software. 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.

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 these accidents, 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. The test and simulation processes must reflect the environment accurately. Although following this process 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 is the use of Ariane 4 trajectory data in the specifications and simulations of the Ariane 5 software when the Ariane 5 trajectory was known to be different.

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 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 engineers incorrectly thought the rigid-body simulation of the vehicle would not exercise the filters sufficiently. Even the tests performed on the Titan 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.

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. Another common problem with testing, particularly software testing, is inadequate emphasis on off-nominal and stress testing.

Deficiencies in Safety-Related Information Collection and Use

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 Challenger report noted that what had once been an outstanding hazard information system had broken down amid cost-cutting and complacency. The situation at NASA today does not appear to be any different. 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 mishap, the use of voice mail and email implies there either was no formal anomaly reporting and tracking system (none is mentioned in the report) 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 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) implemented an informal email process for reporting anomalies and safety concerns or issues that reportedly was highly successful.

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.

SUMMARY

Software and digital systems provide tremendous power in building complex systems not previously possible. But this increase in power comes with a price—large software systems are fiendishly difficult to get correct. The difficulty of building such software is often underestimated by engineers. To avoid future losses, we need to take a realistic view of the risk accompanying the use of software and develop and use extensions to standard system and safety engineering techniques to handle it.

REFERENCES

1. J.C. Knight and N.G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. *IEEE Trans. on Software Engineering*, SE-12:1, Jan. 1986.

2. N.G. Leveson. *Safeware: System Safety and Computers.* Addison Wesley, 1985.

3. N.G. Leveson. Intent Specifications: An Approach to Building Human-Centered Specifications. *IEEE Trans. on Software Engineering*, SE-26:1, Jan. 2000.

4. D.A. Mackall. Development and Flight Test Experiences with a Flight-Critical Digital Control System. NASA Technical Paper 2857, Dryden Flight Research Facility, Nov. 1988.