Software Challenges in Achieving Space Safety[1]

Nancy G. Leveson
Aeronautics and Astronautics Department
Massachusetts Institute of Technology
77 Massachusetts Ave.
Cambridge, MA 02142, USA
leveson@mit.edu

Summary: Techniques developed for hardware reliability and safety do not work on software-intensive systems; software does not satisfy the assumptions underlying these techniques. The new problems and why the current approaches are not effective for complex, software-intensive systems are first described. Then a new approach to hazard analysis and safety-driven design is presented. Rather than being based on reliability theory, as most current safety engineering techniques are, the new approach builds on system and control theory.

Key Words: Spacecraft safety, software safety, spacecraft software engineering

1. Introduction

Software is being used increasingly to control critical space activities. Not surprisingly, the number of serious incidents and losses attributable to software flaws has also increased accordingly. Two of the U.S. Mars 98 exploration projects were lost due to software flaws, including Mars Polar Lander (MPL) [1,2] and Mars Climate Orbiter (MCO) [3,4]. Most of the other Mars missions had serious software problems after launch but they did not occur in critical functions or occurred at times when they could be repaired from the ground, including software problems in Mars rovers and in Pathfinder. A software error doomed the Mars Global Surveyor spacecraft [5]. In 1999, a Titan IV/Centaur tasked to place a Milstar satellite in geosynchronous orbit, because of a software problem, placed the satellite in an incorrect and unusable low elliptical final orbit and it had to be destroyed [6]. SOHO (Solar Heliospheric Observatory) was a joint effort between NASA and ESA to perform helioseismology and to monitor the solar atmosphere, corona, and wind. Following a successful period of nominal activity, contact with SOHO was lost in June, 1998, after a series of errors in making software changes along with errors in performing a calibration and momentum management maneuver [7]. The first flight of the Ariane 5 was lost due to software errors [8]. Cassini/Huygens had a potentially serious communications disruption due to software flaws [9]. Most spacecraft engineers and space scientists can recount such stories. This paper describes the special challenges to safety in using software in spacecraft.

Safety here is defined broadly, as is common in the defense and some parts of the aerospace community. While some engineers associate accidents (and thus safety) only with death or injury, accidents are more generally defined as undesired and unplanned events that result in a loss, including loss of human life or injury, equipment, mission, and so on [for example, 10,11]. Certainly the loss of a billion dollar spacecraft that has taken a decade to develop is a significant loss that is worth expending some energy to prevent and thus falls within the definition of an accident. Safety engineering is used to prevent accidents or losses. The surprising fact is that although many regard safety engineering as increasing the cost of system development, if safety analysis is used to assist in making the early design decisions, the cost to build a safer system need be no greater than normal development.

---

[1] This paper is to appear in the Journal of the British Interplanetary Society, special issue on space safety.

Understanding the Problem

The reason safety and reliability cost so much is that they often are considered only after the major architectural design decisions have been made. At that point, the only choice is to add expensive redundancy or excessive design margins. It has been estimated in the defense community that 70-80% of the decisions affecting safety are made in the early concept development stages of a project [12]. Compensating later for making poor choices at the beginning can be very costly.

Those not familiar with software engineering and the difficulties involved sometimes assume that the use of digital components and software will automatically improve reliability and safety over analog devices and electro-mechanical systems because the failure modes of the old technology are eliminated or reduced by using software. In fact, complacency and the discounting of the risks created by software has been a recurring causal factor in most of the software-related spacecraft losses. For example, in the SOHO loss, overconfidence and complacency, according to the accident report, led among other things to inadequate testing and review of changes to ground-issued software commands to the spacecraft [7]. Protections built into the process, such as the review of critical decisions, were bypassed. Even 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 Ariane 5 accident report notes that software was assumed to be correct until it was shown to be faulty [8]. The opposite assumption is more realistic. Engineers often underestimate the complexity of software and overestimate the effectiveness of testing [13].

While it is true that older failure modes are eliminated by using software, at the same time software introduces new types of flaws, some of which are even more difficult and expensive to combat than the old ones they replaced. That does not mean that software and digital systems should not be used—they provide the ability to perform functions and achieve goals that were not practical in the past. It does mean, however, that using software will not automatically eliminate potential problems—special care and new procedures are required to avoid more software-related spacecraft losses like those we have been experiencing (and need not have had). Determining what new procedures are needed requires first understanding the new problems that are introduced by software.

Most engineering techniques for increasing reliability depend on redundancy and overdesign (building in safety margins). Unfortunately, none of these techniques apply to software and some of the losses experienced lately have resulted from misguided attempts to apply them or rely on them. The maiden flight of the Ariane 5 is an example. About forty seconds after initiation of the flight sequence, at an altitude of 2700 meters, the launcher veered off its flight path after it lost all guidance and attitude information, broke up, and exploded. The accident report notes that only random failures were addressed in the Ariane program, and they were primarily handled with redundancy [8]. In the accident, both the primary and backup inertial reference system computers shut themselves down due to a common design error (the same software was used in both computers). Obviously, redundant copies of flawed logic do not provide any protection against those flaws.

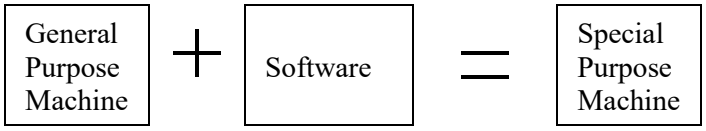
To cope with software design errors, “diversity” has been suggested in the form of independent groups writing multiple versions of software with majority voting on the outputs (like modular redundancy in hardware). This approach is based on the assumption that such versions will fail in a statistically independent manner, but this assumption has been shown to be false in practice and to be ineffective in both carefully controlled experiments and mathematical analysis [14,15,16]. Common-cause (but usually different) logic errors tend to lead to incorrect results when the various software versions attempt to handle the same unusual or difficult-to-handle inputs. The lack of independence in the multiple versions should not be surprising as human designers do not make random mistakes; software engineers are not just monkeys typing on typewriters. As a result, versions of the same software (derived from the same requirements)

developed by different people or groups are very likely to have common failure modes—in this case, common design errors.

In addition, such redundancy schemes 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 the software problems occurring during flight testing resulted from errors in the redundancy management system (which was necessarily much more complex than the original control software). The control software versions worked perfectly [17].

The most important limitation of this technique stems from the fact that most software-related accidents result from requirements deficiencies, usually incompleteness [18]: Multiple versions of software written from the same incomplete or incorrect requirements specification obviously will not prevent an accident stemming from flaws in the specification.

Some of the confusion arises from the wide-spread but untrue belief that safety and reliability are equivalent. In relatively simple hardware systems, most of the design errors can be removed through testing, so the problems remaining after development primarily stem from wear out and other types of component failure. As a convenient label, let's call these *component failure accidents*. The same is not true for software, however. In fact, software does not wear out or fail randomly like hardware. The uniqueness and power of the digital computer over other machines stems from the fact that, for the first time, we have a general purpose machine:



We no longer need to build a mechanical or analog attitude control system from scratch, for example, but simply to write down the "design" of the attitude control system in the form of instructions or steps to accomplish the desired goals. These instructions are then loaded into the computer, which, while executing the instructions, in effect *becomes* the special purpose machine (the attitude controller). If changes are needed, the instructions can be changed instead of building a different physical machine from scratch. Software then is the *design of a machine abstracted from its physical realization*. What does it mean for an abstraction to fail? It certainly does not wear out or suffer physical degradation over time. Computer hardware is now very reliable and almost all the problems associated with computers stem from confusion by the designers about what the software should do.

While software design errors should theoretically be detectable by examination and testing prior to use, software allows the creation of enormously complex designs where the errors are almost impossible to detect and remove. In addition, the problems usually are not in coding, as noted above, but in the requirements. The result is that software-related accidents involve a new type of accident, which can be called a *component interaction accident*: None of the components fail (all satisfy their specified requirements) but the problems arise from dysfunctional interactions among the components. The loss of the Mars Polar Lander was attributed to noise (spurious signals) generated when the landing legs were deployed during descent [1,2]. This noise was normal and expected and did not represent a failure in the landing leg system. The onboard software interpreted these signals as an indication that landing occurred (which the software engineers were told they would indicate) and shut the engines down prematurely, causing the spacecraft to crash into the Mars surface. The landing legs and the software performed correctly (as specified in their requirements, i.e., neither failed), but the accident occurred because the system designers did not account for all interactions between the leg deployment and the descent-engine control software.

Finding these errors before launch is very difficult. As an example, the model we built for the FAA of the software logic in TCAS, a collision avoidance system required on most commercial aircraft, contained 10^{40} states [19]. The TCAS logic is less complex than that used in most

spacecraft today. While some formal methods, like model checking are potentially possible, they are limited in the types of errors they can find and, even then, are very expensive to use.

All of these problems are compounded by the fact that the current widely used safety and reliability engineering analysis techniques, such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA), were designed to identify component failure accidents and thus do not apply to software and to component interaction accidents. The attempts to force fit software into these techniques are like trying to fit a square peg in a round hole: The basic assumptions about accident causes underlying FTA and FMEA do not hold for software [20]. Often hazard analyses simply omit software, and when included it is usually 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 [2].

The problem with such vacuous statements about software hazards is that they provide no useful information—they are equivalent to substituting the general statement *Hardware fails to operate properly* for all the other identified system hazards. What can engineers do with such 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 by the author in 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. The problem faced by engineers is that they have no other, more appropriate, tools to use.

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 of 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*. The problem then reduces to finding new approaches to hazard analysis and design for safety that help to identify these types of dangerous behavior early in development so they can be used to prevent software-related accidents.

2. A New Accident Model and Hazard Analysis Approach

All of this seems pretty grim. If our standard techniques such as redundancy, safety margins, and analysis techniques such as FMEA and Fault Trees do not work for software-intensive systems, what can we do instead?

There is a way forward and that is to expand the accident causality models used in spacecraft design to include the new types of accident causes created by the use of software. One way of doing this has been proposed [21] where control theory and system theory are substituted for the current reliability theory underlying current techniques. In this new model of accident causality, called STAMP (Systems-Theoretic Accident Model and Processes), accidents are viewed as a control problem and result from a lack of enforcement of safety constraints on the behavior of the system. Safety is defined in terms of constraints on the behavior of the system components. The safety constraint violated in the Mars Polar Lander loss, for example, is that the spacecraft does not impact the planet's surface with greater than a specific amount of force.

STAMP includes traditional component failure accidents—the hazards result from inadequate control of component failures and their consequences—but it also includes component interaction accidents where the system does not control unsafe interactions of components, such as insulation falling off the external fuel tank (after it had performed its required function) and damaging the spacecraft thermal protection system (Columbia) or software misinterpreting signals from the landing legs and shutting down the descent engines prematurely (MPL). In Challenger, the problem was not just the failure of the O-ring, but more generally that the O-ring did not control propellant gas release by sealing the gap in the field joint: The O-ring was a feature designed to control a hazard (propellant gas release through the field joint).

Starting with high-level safety constraints at the beginning of spacecraft design, a new form of hazard analysis (called STPA) can be used to refine the safety constraints and spacecraft design decisions in parallel in a process called *safety-driven design* [22,23]. Hazards (potential causes of the loss of the spacecraft) are eliminated, if possible, as design decisions are made. If the hazards cannot be eliminated, then decisions must be made about how to mitigate or control them in the design.

Hazards can arise either because of component failures (for hardware) or because of unsafe interactions among the spacecraft components (for hardware and software). Both of these need to be considered, not just component failures. Note that all the techniques, such as redundancy and overdesign, used currently to implement fault tolerance may still be used. It is just that additional new design features that apply to software and to hardware design faults are identified or suggested through the process.

Because safety is treated as a control problem, STPA works on control diagrams. Figure 1 shows an example of the functional control structure for an outer planets explorer spacecraft generated in a demonstration of the process for NASA JPL [23]. The control structure was generated by a process that uses the high-level requirements and constraints, including safety constraints, to minimize interactions among the system components.

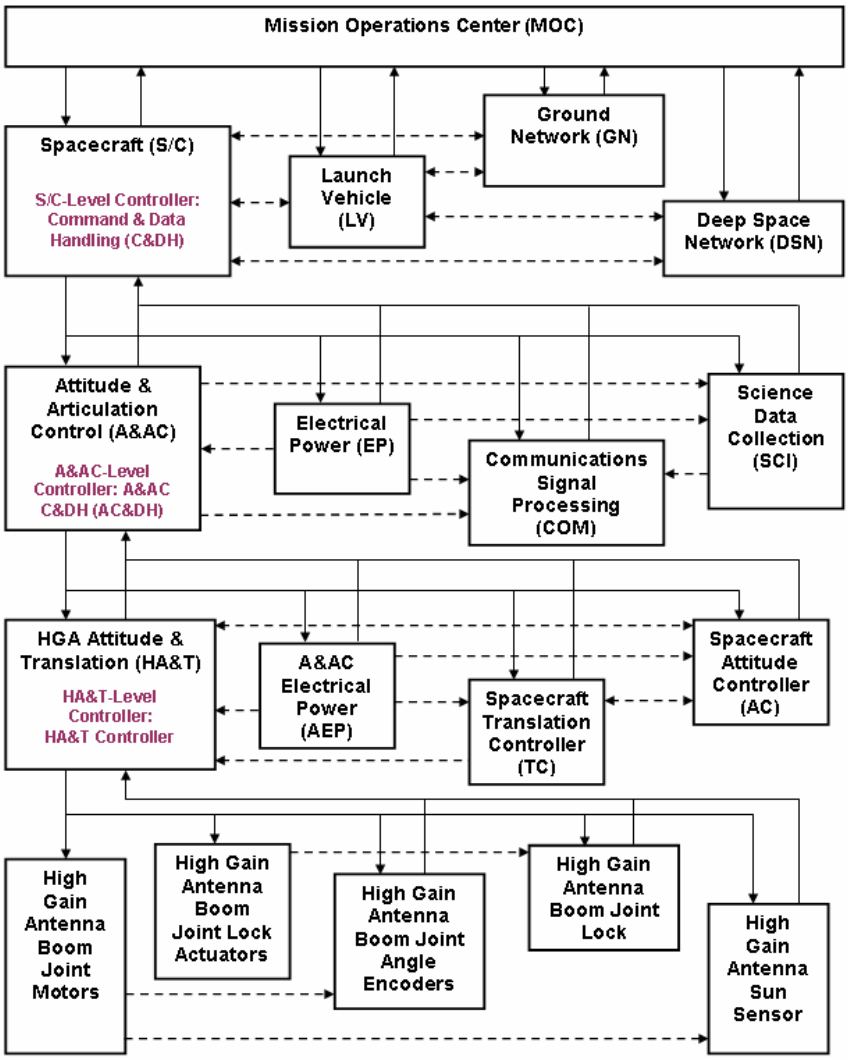
As the control loops and algorithms implemented by the controllers are refined, STPA is applied to them to identify the safety constraints that must be enforced by the control loop and identify the ways in which control can be inadequate. Note that the process is similar to the more traditional fault tree analysis, but control theory concepts are used to consider more than just failures, the process can be used before any design exists, and more guidance is provided to the analyst.

Using this new control theory model of accidents as inadequate enforcement of safety constraints, we can identify the general ways that safety constraints may not be enforced:

1. The controller issues inadequate or inappropriate control actions, including inadequate handling of failures or disturbances in the physical process or
2. The control actions are inadequately executed by the actuators.

Refining the first cause listed above, losses may occur because

1. The controller issues an unsafe control action.
2. The controller does not issue a control action needed to enforce safety
3. The controller issues the necessary control action but not at the right time (too early or too late).
4. The control action is stopped too soon or too late.



Control Structure Legend	
Diagram Item:	Description:
↓	Control in the form of Directive(s) or Command(s)
↑	Control Feedback in the form of State Information or Sensor Measurements
↔	Physical and Informational Interaction other than Control and Control Feedback Interactions
Functional Element Name Functional Element-Level Controller (if applicable)	Functional Element with the controller of its internal interactions (i.e., the functional element-level interactions)

Fig. 1: The Control Structure for an Outer Planets Explorer Spacecraft []

Figure 2 shows a basic control loop and the reasons why the control loop might not enforce the safety constraints. These causal factors are used in STPA to identify accident scenarios, which can then be used to make design decisions about how to eliminate or control the impact of the scenarios. The factors labeled 1, 2, and 3 are related to inadequate control actions while those labeled 4 involve inadequate execution of the provided control actions.

While most of the figure should be self-explanatory, there are a couple of things that need further explanation. Engineering techniques often assume systems are static, but in reality they tend to change over time. Hazards can be created through these changes.

In addition, the concept of a process model is new to most hazard analysis techniques but critical when considering software or human controllers. Every controller uses an internal model of the controlled process in order to create control commands. This internal model can be called the Process Model (labeled 3 in the figure). Accidents often occur when the internal process model becomes inconsistent with the real state of the controlled process. For example, the internal model of the software controlling the descent engines on MPL did not match the state of the spacecraft—the software thought the spacecraft had landed and shut down the descent engines prematurely.

The ways the process model can become inconsistent with the actual state of the controlled component must be identified in the hazard analysis process and then eliminated or controlled in the design of the component, the controller, or the spacecraft as a whole. Clearly, there is no one way to solve the problem, engineers must consider the tradeoffs between the various protection mechanisms and algorithmic solutions. One possible solution in this example might be to include an altimeter that provides additional information to the software about the state of the spacecraft. There are many other better or more practical solutions involving the design of the software and the design of the spacecraft. In many cases, simply identifying the potential cause of the hazard allows designing the software to prevent it.

Just as with any hazard analysis technique, STPA helps the engineer identify the potential causes of violating the safety constraint so they can be eliminated or controlled. It includes, however, a broader set of causal factors than usually considered. In addition, it can be started before any design decisions have been made (beyond identifying the basic mission of the spacecraft and the general functions needed to successfully perform the mission) and used to guide the design decisions as they are made instead of waiting until a design has been proposed and then analyzing that detailed design for loss scenarios. STPA and the process of safety-driven design are described more completely and examples provided in [20].

Because STPA is new, a legitimate concern is whether it works. While no controlled experiments have been performed, experience in practice is promising. Only the aerospace examples are included here although it is being applied in other industries. STPA was used to perform a non-advocate safety analysis of the new U.S. missile defense system before its deployment and testing [24]. This new system is extremely complex and made up of a large number of components, some of which are new and some (such as early warning and radar systems) have been operational for decades. The hazard considered was inadvertent launch. STPA found a large number of scenarios not found by the traditional techniques—in fact, so many were found that deployment and testing were delayed for six months while fixes were identified and implemented. Many of the new identified scenarios were, as predicted, in the interactions among the system components and did not involve individual component failures. Component failure scenarios were also identified. As a result of the success of this assessment, STPA has been adopted by the U.S. Missile Defense Agency as their primary hazard analysis approach. It is also being used on Orion, the replacement for the Space Shuttle. Finally, a demonstration project on an outer planet explorer spacecraft design has been completed for NASA JPL [23] and a demonstration on a JAXA spacecraft is just beginning.

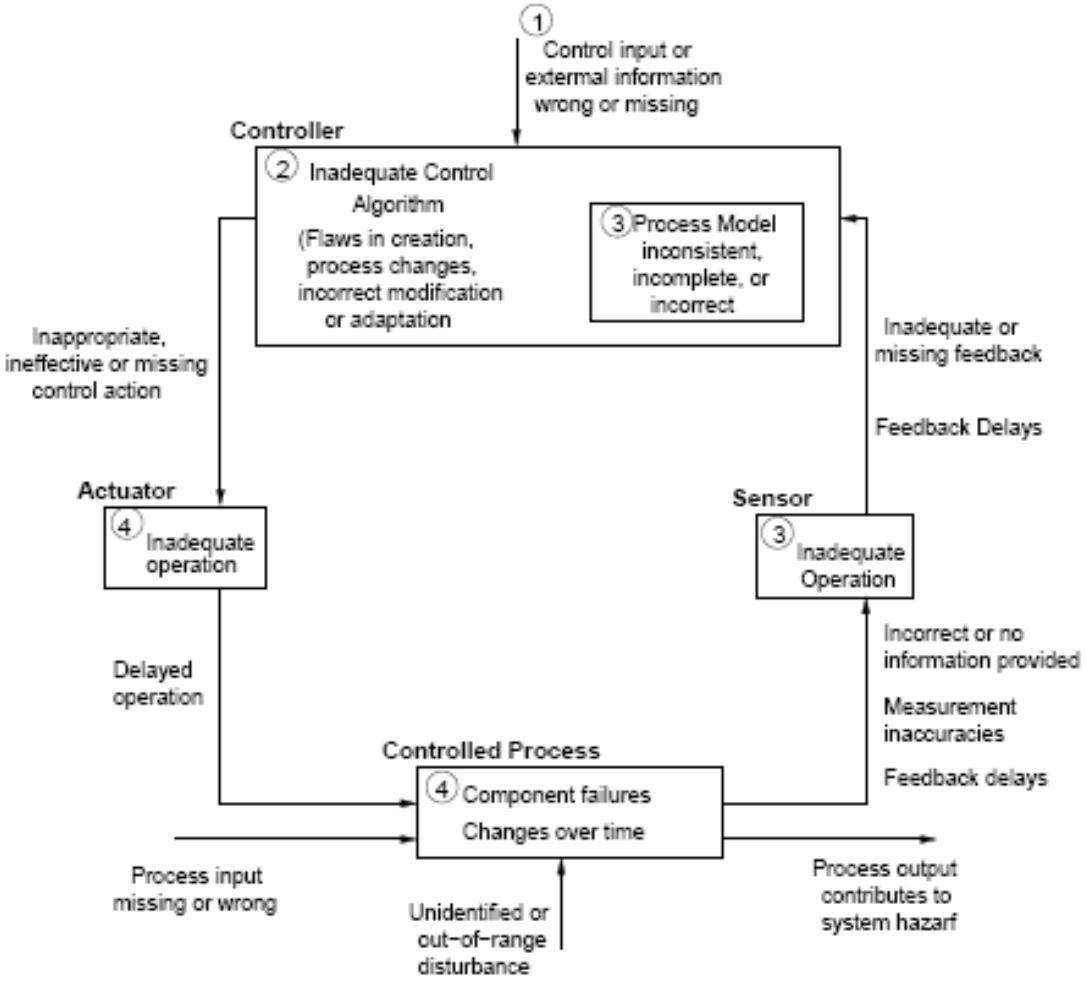


Fig. 2: The Causal Factors in Spacecraft Losses

3. Other Components of the Solution

While an expanded model of accident causation and new hazard analysis and design techniques are important components of dealing with the new challenges of software-intensive spacecraft design, other changes or new emphases in spacecraft development are also required.

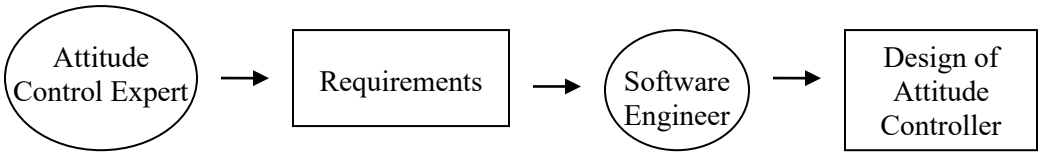
3.1 System Engineering

Because of the increased complexity that the use of software allows in spacecraft design, system engineering and sophisticated system engineering tools assume an even greater importance than for purely analog designs. Software allows us to build systems of such high interactive complexity that it is difficult, and sometimes impossible, to plan, understand, anticipate, and guard against potential unintended interactions.

For any project as complex as those involved in these accidents, good system engineering is essential for success. Preventing component interaction accidents cannot be the responsibility of those designing the individual spacecraft components, who have little control over events arising

from unsafe interactions with other components. These potential interaction problems must be identified and handled at the spacecraft level by system engineering.

Software development is often isolated from the rest of project—it is treated as simply a subsystem component like any other. But software controls the interactions among components and plays a coordinating role among the system components. It cannot be designed in isolation from them and from the system engineering activities. In fact, software engineers are not just doing software design, they are doing system and spacecraft design. In Section 2, software was described as the design of a system abstracted from the physical realization of that design. The attitude control expert, for example, provides functional requirements to the software engineer who actually designs the attitude controller.



But the software engineer, who usually has some knowledge about spacecraft design, may not be an expert in attitude control. It is not surprising that almost all accidents related to software stem from problems in the requirements specification, particularly incompleteness [18,25]. The software designer has to assume that those things not in the requirements specification are irrelevant with respect to safety (and mission accomplishment). Requirements are the key here, and system engineering guides the process of producing those requirements.

In some of the recent spacecraft accidents, system engineering resources were insufficient to meet the needs of the project. In others, the process followed was flawed, such as the flowdown of system requirements to software requirements (MPL) or in the coordination and communication among project partners and teams. In the Titan/Milstar 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 system engineering process [6]. 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 or prevent the software errors that occurred.

3.2 Specification Practices

The vast majority of software-related accidents have been related to flawed requirements and misunderstanding about what the software should do. Software-related spacecraft accidents often refer to inadequate specification practices. The Ariane 5 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 [8]. Inadequate documentation of design rationale to allow effective review of design decisions is a common flaw in system and software specifications [3]. The Ariane 5 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 accident report notes that the system-level requirements documentation did not specifically state the failure modes the requirement was protecting against (in this case, transient signals from the landing leg sensors) 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 [2]. In addition, standards and industry practices often forbid "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. Although there are differing accounts about what happened with MCO, all accounts agree that the specifications were flawed [26].

Complete and understandable specifications are not only necessary for development; they are also critical for operations and the handoff between developers, maintainers, and operators. In the Titan/Milstar loss, 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 [6]. As a consequence, when discrepancies were discovered right before launch, nobody understood them. The MCO operations staff also clearly had inadequate understanding of the automation, and they were unable to monitor its operation effectively [3]. 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 [7]. The report also states that the missing software *enable* command leading 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. Safety-critical information 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 [9], the designs or parts of designs are labeled as "proprietary" and cannot be reviewed by those who could provide the most important input. Adequate system engineering is not possible when the system engineers do not have access to the complete design of the spacecraft. As argued repeatedly in this paper, the software contains the design of the spacecraft—software cannot just be treated as a black box component.

3.3 Complexity and Software Functionality

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 pushes this basic principle into the background: *Creeping featurism* is a common problem in software-intensive systems:

> "And they looked upon the software, and saw that it was good. But they just had to add this one other feature' . . . A project's specification rapidly becomes a wish list. Additions to the list encounter little or no resistance. We can always justify one more feature, one more mode, one more gee-whiz capability. And don't worry, it'll be easy—after all, it's just software. We can do anything. In one stroke, we are free of nature's constraints. This freedom is software's main attraction, but unbounded freedom lies at the heart of all software difficulty." (Frank McCormick, unpublished essay).

Many spacecraft losses have involved either unnecessary software functions or software operating when it was not necessary. Both the Ariane and Titan/Centaur accidents involved software functions that were not needed, but surprisingly the decision to put in 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 liftoff.

The Ariane 5 accident report questions the advisability of retaining the unused Ariane 4 alignment function in the Ariane 5 software, but it does *not* question whether the Ariane 4 software should have included such a non-essential software function in the first place. Outside of its effect on reuse (which may reasonably not have been contemplated during Ariane 4 development), a tradeoff was made between possibly delaying a launch and simplifying the software. Given that the function was used only once in the history of the Ariane 4 program and only averted a potential launch delay and not a critical function, the decision to include it seems questionable.

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 report states that the decision to start the software early was based on trying to avoid transients in CPU loading. The tradeoff in this case may have seemed justified, but executing software when it is not needed or including unnecessary code raises risk significantly. The MPL independent accident investigation report also noted that requirements creep was a factor in the accident [1,2].

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, the tradeoffs between safety and added complexity 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. Well-reasoned tradeoff decisions must include a realistic assessment of the impact of these extra difficulties. Software functions are not “free.”

3.4 Problem-Reporting

Several losses have resulted from reporting system flaws in handling anomalies detected in the software during operations. In the Titan/Centaur [6] and Mars Climate Orbiter [3,4] accidents, for example, there was evidence that a problem existed in the software 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. Engineers noticed the problems with the Titan/Centaur software, for example, after it was delivered to the launch site, but nobody was assigned responsibility for detecting or handling software problems detected during launch operations. No communication channel was established for getting the information to those who could understand it. In some losses, the reporting channel existed but was ineffective in some way or was simply unused.

In other cases, the problem reporting channel was operational, but the response to problem reports was inadequate. A deficiency report on 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 software during the early stages of flight did not seem to raise any red flags.

3.5 Software reuse

Reuse and the use of commercial off-the-shelf software (COTS) is common practice today in embedded software development and reuse was implicated in most of the accidents cited in this paper. The Ariane 5 software involved in the loss was reused from the Ariane 4, as noted earlier. According to the MCO developers, the software function implicated in that loss was reused from the Mars Global Surveyor (MGS) project, with the substitution of one new equation [4]. Technical management accepted the "just like MGS" argument and did not focus on the details of the software. The SOHO software was changed without appropriate analysis of the changes [7].

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: 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 black box (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.*

The problems in the Ariane 5 software arose from an overflow of a 16-bit integer variable. The horizontal velocity of the Ariane 4 cannot reach a value beyond the limit of the software, i.e., the value will always fit in 16 bits. The initial acceleration and trajectory of the Ariane 5, however, leads to a buildup of horizontal velocity five times more rapid than that of the Ariane 4, leading to an overflow. For some unexplained reason, the Ariane 5 requirements and specifications did not include the Ariane 5 trajectory data. This omission was not simply an oversight—the accident report states that the partners jointly agreed not to include it [8]. The report provides no reason for this very strange decision, which negatively impacted the ability to detect the problem during reviews and testing.

The philosophy of the Ariane 5 program, as stated in the accident report, that it was not wise to change software that had worked well on the Ariane 4 unless it was proven necessary to do so is well founded: Errors are often introduced when software is changed, particularly by those who did not originally write it, as occurred with SOHO. However, in this case, there is a tradeoff with safety that needs to be carefully evaluated. The best solution may not have been to leave the software as it was but to rewrite it from scratch—such a decision depends on the type of change required, the specific design of the software, and the potential effect of the feature on system safety. The cost of the decision to reuse the Ariane 4 software without rewriting it (in terms of both money and program delays) was much greater than the cost of doing a proper analysis of its safety. The same is true for the Centaur software with respect to the losses resulting from leaving in the unneeded filter function. If the cost of the analysis of reused (or COTS) software or of changes to software is prohibitively expensive or beyond the state of the art, then redeveloping the software or completely rewriting it may be the appropriate decision.

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.

A similar argument applies to changing existing software. Changes to software appear to be easy. While it is indeed easy to change software, it is very difficult to change it correctly and the difficulty increases over time. The Mars Climate Orbiter software was changed to include a new thruster equation, but the 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 [4]. Modifications to the SOHO command procedures were subjected to very little testing and review, perhaps because they were considered to be minor [7].

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. 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.

3.6 Human-Computer Interaction

Understanding the impact of software design on human error is still in the early stages. Accidents, surveys and simulator studies, however, have emphasized the problems pilots are having in understanding digital automation and have shown that pilots are surprisingly uninformed about how the automation works [27]. Designers of glass cockpit and highly automated aircraft have been grappling with the new risks that have been introduced.

As more sophisticated automation is used in spacecraft control, and the control of safety-critical functions is increasingly shared between humans and computers, the problems found in high-tech aircraft are appearing. The need for an accurate process model (called a mental model in humans) noted in Section 3 is just as necessary for a human controller as an automated controller. For humans who are interacting with computers, the operator must not only have an accurate model of the controlled system but also of the state of the computer and how it works. Neither the MCO 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. 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 computer interface provided.

Complexity in the automation combined with poor documentation and training procedures are contributing to these problems. 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. As a result, they were unable to detect the software deficiencies in time to save the mission.

3.7 Conclusions

The use of software in spacecraft design provides many important advantages and functions that might be impractical without it. But it also presents important new challenges and potential for accidents. System safety techniques and other spacecraft engineering activities need to be expanded to include both software and the complex, cognitive decision-making roles played by human operators.

References

1.Young, T. (Chairman), "Mars Program Independent Assessment Team Report," NASA, 14 March, 2000.
2. 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.
3. Stephenson, A., "Mars Climate Orbiter: Mishap Investigation Board Report," NASA, November 10, 1999.
4. 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," *Proceedings of Guidance and Control 2001*, American Astronautical Society, paper AAS 01-074, 2001.
5. Perkins, Dolly. NASA Internal Review Board Report on Mars Globarl Surveyor (MGS) Spacecraft Loss of Contact, April 13, 2007.
6. 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.
7. NASA/ESA Investigation Board, "SOHO Mission Interruption," NASA, 31 August 1998
8. Lions, J.L. (Chairman) "Ariane 501 Failure: Report by the Inquiry Board," European Space Agency, 19 July, 1996.
9. D.C.R Link (Chairman), "Report of the Huygens Communications System Inquiry Board," NASA, December 2000.
10. U.S. Department of Defense. Standard Practice for System Safety: MIL-STD-882D, February 2000.
11. NASA, NRP 8715.3: General Program Safety Requirements, April 17, 2009.
12. Frola, F.R. and Miller, C.O., "System Safety in Aircraft Acquisition," Logistics Management Institute, Washington D.C., January 1984.
13. Leveson, Nancy G. "The Role of Software in Spacecraft Accidents," *AIAA Journal of Spacecraft and Rockets*, **41**, No. 4, July 2004.
14. Knight, J.C. and Leveson, N.G., "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Transactions on Software Engineering*, **SE-12**, No. 1, January 1986, pp. 96-109.
15. D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", *IEEE Trans. on Software Engineering*, **SE-11**, No. 12, December 1985, pp. 1511-1516.
16. Knight, J.C. and Leveson, N.G. A Reply to the Criticisms of the Knight and Leveson Experiment, *ACM Software Engineering Notes*, January 1990.
17. Mackall, D.A., "Development and Flight Test Experiences with a Flight-Critical Digital Control System," NASA Technical Paper 2857, Dryden Flight Research Facility, November 1988.
18. Leveson, N.G., *Safeware: System Safety and Computers*, Addison Wesley, Boston, 1985.
19. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., and Reese, J.D. "Requirements

Specification for Process-Control Systems," *IEEE Transactions on Software Engineering,,* **SE-20**, No. 9, September, 1994.
20. Leveson, N.G. Engineering a Safer World: Applying Systems Thinking to Safety. Expected, MIT Press, 2010. Draft copy available at http://sunnyday.mit.edu/book2.pdf.
21. Leveson, N.G. "A New Accident Model for Engineering Safer Systems," *Safety Science*, **42**, No. 4, April 2004, pp. 237-270.
22. Stringfellow, Margaret V. (2008), *Safety-Driven System Engineering Process*, S.M. Thesis, Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA.
23. Brandon Owens, Margaret Herring, Nancy Leveson, Michel Ingham, Kathryn Weiss, "Application of a Safety-Driven Design Methodology to an Outer Planet Exploration Mission," *2008 IEEE Aerospace Conference*, Big Sky, Montana, March 2008.
24. Pereira, Steven J., Grady Lee, and Jeffrey Howard. "A System-Theoretic Hazard Analysis Methodology for a Non-advocate Safety Assessment of the Ballistic Missile Defense System," *Proceedings of the 2006 AIAA Missile Sciences Conference***,** Monterey, California, November 2006.
25.Lutz, R.R., "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the International Conference on Software Requirements*, IEEE, January 1992, pp. 53-65.
26. Oberg, J., "Why the Mars Probe Went Off Course," *IEEE Spectrum Magazine*, **36**, No. 12, December 1999.
27. Bureau of Air Safety Investigation, "Advanced Technology Aircraft Safety Survey Report," Department of Transport and Regional Development, Australia, June 1996.