The Natural History of Bugs: Using Formal Methods to Analyse Software Related Failures in Space Missions

C.W. Johnson,

Department of Computing Science, University of Glasgow, Glasgow, G12 9QQ.
johnson@dcs.gla.ac.uk

Abstract. Space missions force engineers to make complex trade-offs between many different constraints including cost, mass, power, functionality and reliability. These constraints create a continual need to innovate. Many advances rely upon software, for instance to control and monitor the next generation 'electron cyclotron resonance' ion-drives for deep space missions. Programmers face numerous challenges. It is extremely difficult to conduct valid ground-based tests for the code used in space missions. Abstract models and simulations of satellites can be misleading. These issues are compounded by the use of 'band-aid' software to fix design mistakes and compromises in other aspects of space systems engineering. Programmers must often re-code missions in flight. This introduces considerable risks. It should, therefore, not be a surprise that so many space missions fail to achieve their objectives. The costs of failure are considerable. Small launch vehicles, such as the U.S. Pegasus system, cost around $18 million. Payloads range from $4 million up to $1 billion for security related satellites. These costs do not include consequent business losses. In 2005, Intelsat wrote off $73 million from the failure of a single uninsured satellite. It is clearly important that we learn as much as possible from those failures that do occur. The following pages examine the roles that formal methods might play in the analysis of software failures in space missions.

The Challenges of Software Engineering in Space

Space is unforgiving. The following sections briefly review some of the challenges that complicate software development in this environment.

The Usual Suspects

'Rocket science' is often seen as the pinnacle of scientific and technological progress. For instance, it has been estimated that there are more than 1.5 million lines of code in the onboard command and control computers on the International Space Station. However, such figures are commonplace in several other industries. The day-to-day reality of maintaining space-related code would also be familiar to other software

© Springer-Verlag, http://www.springer.de/comp/lncs/index.html

engineers. For example, the Expedition 10 crew is on the International Space Station as I write this article. Part of their six-month stay will be used to install software upgrades. These are intended to eliminate the 300 workarounds, 'Station Program Notes', that are used by ground flight controllers [15].

The causes of many failures in space missions will also be familiar to software engineers. These include the under-specification of complex systems, lack of resources for validation and verification, poor communication between multidisciplinary teams and so on. One consequence of this is that many academic computer scientists cite software failures from space missions as warnings to their students about what can go wrong in their own programs. The most familliar examples include the Ariane 5 code re-use [14] and the confusion over metric and imperial units of thrust in the Mars Climate Orbiter [16]. In contrast, the following pages delve a little more deeply into the challenges that distinguish software engineering for space systems from a mass of other applications.

The Remoteness of Space

One of the first issues to confront a programmer is that many space missions must travel thousands of miles from Earth. This creates a peculiar form of batch processing where code will not be executed until months or even years after launch. Further complexity is created by the possibility of reprogramming these missions in flight. Such reprogramming is widely acknowledged to be both difficult and error prone. For example, some telemetry configurations may not enable programmers to verify that a spacecraft has successfully received instruction sequences. In other words, the target machines are often 'write-only'.

There are significant pressures associated with recoding a space mission as it travels towards a rendez-vous with a distant planet. In consequence, programming teams will often develop coding strategies to reduce the chances for an error. One technique is to program a range of different options before launch. Once the mission is in flight, the team accepts self-imposed limits on the admissible reprogramming that may be attempted. Often the choice will be restricted to one of the pre-scripted instruction sequences planned and loaded before launch [19]. Other missions have adopted hybrid strategies where programmers can only upload new code after multiple reviews and at a small number of key stages in the mission. At all other times, they must rely on prescripted command sequences.

The differences that physical distance impose on the programming of space missions can be illustrated by events involving NASA's Spirit and Opportunity Mars Rovers during September 2004 [20]. Programmers had to transfer Spirit and Opportunity back from 'conjunction' to normal mode. During a conjunction, communications are disrupted because Mars and Earth are on opposite sides of the Sun. During the conjunction, pre-loaded command sequences were used to perform daily science missions, for instance using a Mössbauer spectrometer and a magnet array to analyze dust particles. The Rovers transmitted the data from these experiments to the Mars

Odyssey orbiter. Odyssey then retransmitted the data back to Earth each afternoon. This link was extremely error prone. This created a bottleneck that reduced Spirit's memory available for science data storage from approximately 400 to 100 megabits. The problems were compounded when the mission team began to transmit 'no operation' commands to test direct communications with the Rovers during the conjunction. One of these commands triggered a software 'reset' on Opportunity.

Reprogramming arguably offers greatest benefits to programmers when they correct for problems with their own code. For example, Spirit and Opportunity had to be reprogrammed shortly after they landed on Mars in January 2004 and have been reprogrammed many times since. Spirit suffered a software fault during its navigation of the 'Columbia Hills'. The flight software team identified that an error occurred within a 3-microsecond window of vulnerability when a 'write' command was permitted and attempted on a 'write-protected' area of RAM [20]. The error was subsequently corrected in a software upgrade that was also communicated to Opportunity. Significant changes have been made to their code in order to extend their mission life beyond Summer 2004. For example, Spirit was commanded to avoid using a faulty brake relay on its steering motor. Both Rovers have been reprogrammed to alternate their drive direction to maintain the long-term health of their wheel drives.

Similarly, the Solar and Helioscopic Observatory (SOHO) was reprogrammed in-flight to de-spin one of its three gyroscopes. The gyros were identified as a 'life limiting' factor for the mission as a whole. Of course, such benefits carry risks as well. The de-spun 'A' gyroscope was involved in the SOHO mission interruption as controllers tried to work out which one of the three systems was providing reliable information [18].

Non-Standard Hardware

Software development is complicated because many space applications require specialist hardware. As a minimum requirement, processors must be 'radiation hardened'. For example, the RAD 6000 processor has been tested to demonstrate 0.2 errors per year GCR – Galactic Cosmic Ray background [6]. If the radiation exposure is increased to a level similar to the flare events seen in October 1982 and January 1972 then the rate rises to 0.6 errors per flare.

Space programmers are caught between a 'rock and a hard place'. They must understand the unique features of 'rad-hard' processors. They must also cope with reduced tool support. Specialist devices lack the wide range of software development applications that support Commercial Off The Shelf (COTS) processors. The limited market for space rad-hard devices often does not justify the development of computer-aided software engineering tools. The additional validation criteria imposed on space-rated processors can also exacerbate the 'generation gap' between the facilities provided by this hardware compared to COTS processors.

Many of these problems can be illustrated by the General Purpose Computer configuration on the Shuttle. A five processor redundant architecture is used to perform critical guidance, navigation and control functions. However, the detailed analysis and design necessary to approve both the hardware and software on the GPC array prevented any updates to the processors for over fifteen years. During which time it became increasingly difficult to find vendors and suppliers for this technology. These are not isolated comments. For example, the minutes of subsystems groups reveal similar concerns throughout the Shuttle programme. The Extra-Vehicular Activities equipment board looking at the Caution and Warning System has continued to experience difficulties in supplying the "100 pieces of the EEPROM for the CPU board, which are becoming obsolete" [21]. It is difficult to underestimate the consequences of such supply problems. Storage and re-commissioning procedures must often be considered when utilizing stocks that were not initially acquired by the eventual end-user. Similarly, there are significant training issues associated with obsolete and non-standard hardware platforms.

Pilot projects have begun to develop specialist versions of commercial microprocessors. For instance, the US Defence Technology Program has invested over $50 million in providing a space-rated version of the PowerPC 750 processor. The resulting SCS750 processors can reduce the flare error rate from 0.6 per event, cited above, to 0.36 errors per flare. These individual heavy-ion irradiation errors can be detected and mitigated by the SCS750 processor [6]. However, the application of these hybrid platforms is still in its infancy.

Limitations of Re-Use

There is a surprising degree of re-use in other forms of space engineering. For example, the design of the heat shields and the parachutes on the Mars Surveyor missions were based on designs from the Pathfinder missions. This provides important benefits to engineers in the aftermath of a mission failure. Investigators quickly dismissed these subsystems as causes of the Polar Lander loss because "the high degree of heritage to the successful Mars Pathfinder design, fabrication, test, and flight results (suggests) that the failure of an undamaged heat shield is implausible" [19]. These arguments can be based on limited evidence "there was not an extensive qualification program as part of the Pathfinder design phase, the Pathfinder chute did, in fact, work, thus providing at least one successful occurrence".

In contrast, space missions offer limited opportunities for code re-use. The loss of Ariane 5 provided a salient example of the problems that can arise when software is ported between different space missions. It is important to acknowledge some of the reasons why code re-use is difficult. Command and control software is typically used to interface complex sub-systems. Any unidentified interactions between these components will most often be revealed in the form of software failure. Later sections will also describe an increasing trend to introduce 'band-aid' software that is intended to fix design deficiencies or to achieve cost savings in the wider engineering

of space missions. 'Band aid' code necessarily involves bespoke programming because it provides a short-term fix for underlying problems in the design and development of complex systems.

Limitations of Ground-Based Testing

Much of the software used in space missions cannot easily be tested on the ground. For example, no test was made to establish that Ariane 5's Inertial Reference System (SRI) would behave as intended under the countdown and flight time sequence for the expected trajectory. The Lyons investigation found that "for reasons of physical law, it is not feasible to test the SRI as a 'black box' in the flight environment, unless one makes a completely realistic flight test" [14]. It was possible to conduct a limited form of ground testing by injecting simulated accelerometric signals based on predicted flight parameters using a turntable to simulate launcher angular movements. Only in retrospect was it argued, "Had such a test been performed by the supplier or as part of the acceptance test, the failure mechanism would have been exposed".

Similarly, the attempt to deliver two Deep Space 2 high-impact micro-probes into the surface of Mars, went ahead in spite of concerns by mathematical modelers that they could not reliably analyze the potential impact forces acting on the devices. Their concerns were significant because of the problems involved in conducting other forms of testing. The mission validation exercises relied on an incremental build-test strategy. However, most of the communications system was only qualified with non-functioning brass-board and breadboard components. Issues of cost prevented a full impact test. In addition, delays in the schedule meant that a fully functioning probe was only available relatively late in the programme. To employ destructive testing would have involved a delay to the launch window [19].

Limitations of Executable and Abstract Modeling

The problems of software development for space missions are compounded because abstract models and simulations have often proven to be unreliable. It is common practice to enter into an iterative cycle where software is first developed and tested on a satellite or vehicle simulator [10]. The results from these evaluations are then compared with those results that can be obtained from the eventual platform. However, any discrepancies are just as likely to result in changes to the simulator as they are to changes in the command and control software. For example, both NASA and the European Space Agency operated their own simulators of the joint Solar and Helioscopic Observatory (SOHO) mission. During the mission interruption it was realized that the NASA model predicted some of the problems they were experiencing. However, the results could not be replicated for the ESA models; "analysis of the differing simulation results (ESA vs. NASA simulators) was continuing as the timeline execution was in process… this, in itself, was an indirect factor in the failure scenario since the technical support staff were distracted by the on-going simulation evaluation rather than focusing on the recovery efforts" [18].

The simulators had not been maintained with all on-board software changes that had been implemented on the spacecraft.

There has long been a debate in the formal methods communities about whether executable models can provide an appropriate level of abstraction to support reasoning about critical properties of complex software systems. However, there are aspects of space missions that stretch our ability to model interactions even at the highest level of abstraction. The investigation into the Mars Surveyor mission failures concluded that the "large modeling effort, however, may have not been enough to ensure success given the choice in the design phase of some of the system components, such as the propulsion system and the landing Radar, and given some aspects of the design of the Guidance and Control algorithms/software, which resulted in a system that was extremely difficult to model and more sensitive to model errors than it might have been" [19]. For example, the Polar Lander used pulse-width modulation (PWM) for controlling the thrust of the descent engines rather than the more conventional throttle based system. This reduced the costs of the Polar Lander hardware but greatly increased the complexity of software development for the programmers who had to calculate the exact duration of each engine pulse during the descent; "the complexity of the interactions between the feed system, the thrusters, the structure, the Guidance and Control sensors, and the Guidance and Control algorithms that the PWM approach creates, practically dictate that the only way of verifying the system with high confidence is with a full-scale closed-loop test of the system… this was prohibitive from a cost and schedule point of view and it was not done" [19].

Organizational Complexity and 'Band-Aid' Software

The use of software to compensate for the pulse-width modulation on the Polar Lander provides an example of 'band-aid' software. This code is introduced to fix design mistakes and compromises in other aspects of space systems engineering. Software is used to cover over design problems just as some mothers use sticking plasters to cover a host of injuries sustained by their children. Arguably the best example of band-aid software comes from the Mars Climate Orbiter mission. As mentioned previously many software engineers are aware that the probable cause of this mission failure stemmed from the use of Imperial rather than Metric units in the calculation of thrust for the rocket motors during the mission cruise phase. Few software engineers realize that the rockets were fired as part of Angular Momentum Desaturation (AMD) events. The software was called upon far more often than was originally intended, some estimates state that there were 10 to 14 times more AMDs than planned. AMD events were intended to desaturate the momentum that was built up on an internal flywheel. This momentum was, in turn, used to counteract solar induced momentum on an asymmetrical solar array. Previous missions had used symmetrical solar panels. The Climate Orbiter's novel design again reduced hardware costs but created problems because solar induced momentum skewed the cruise trajectory. In this way, the engineering decision to have asymmetrical solar arrays created the need to counteract the 'uneven' effects of solar induced momentum

on the panels. This was done by spinning the flywheel in an equal and opposite direction to the momentum induced on the solar panels. However, the flywheel could only be used until its momentum threatened the stability of the vehicle. In order to desaturate the flywheel, programmers had to perform the complex calculations that controlled the rocket motors [16].

The problems created by band-aid software are increased by the organizational complexity of many space missions. For example, most of the team that worked on the software and hardware development of the Mars Climate Orbiter was transferred to the design of the Mars Polar Lander. The mission staff that then had to operate the Orbiter during its cruise and orbit acquisition phases, therefore, lacked many of the insights that might have been provided by the original coders. In other missions, there are conflicts between the programmers who must maintain the integrity of the platform and those who have a primary interest in particular scientific objectives. For instance, the SOHO Flight Operations Team was encouraged to modify the stored sequences of ground-generated commands. These modifications reduced operational cost during the extended life of the mission; they also minimized science 'downtime' and conserved the gyro life. Some modifications proposed by the Science Team 'were not necessarily driven by any specific requirement changes' [18]. The modifications were not adequately managed, for example not all of them were considered by a Configuration Board. Many were poorly documented. Verification relied on the NASA computer-based simulator, mentioned previously. There were no code walk-throughs, no independent reviews by ESA or any other body not involved in the implementation of the change. No hard copy of the command procedure set on the satellite existed at the time of the mission interruption.

Formal Methods in the Development of Space-Related Software

There have been a number of notable attempts to use formal methods to address the problems of software engineering for space-related applications. SRI have used a range of theorem provers, such as PVS, and model checking tools, including Murϕ to verify that there are no violations of desired properties in models of a system. One of the best-known examples of this work includes the analysis of the software for the Simplified Aid for Extra-Vehicular (EVA) Rescue, known as SAFER. This can be thought of as a form of jet-pack [17]. Other projects have looked at the Shuttle's contingency guidance system [3]. In Europe, the Picgal project has used VDM to analyze ground-based software for launch vehicles similar to Ariane 5 [4]. Relatively slow progress has been made towards the introduction of these techniques as tools for the development of space-related software. One reason for this is the relative immaturity of contemporary software engineering practices in space applications. A number of more basic software engineering processes provide greater benefits at lower costs.

The remainder of this paper looks at an alternate use of formal methods. Rather than focusing on the constructive use of formal methods during program development,

these techniques can be used to help us analyze the causes of software failures in space missions.

Understanding Space-Related Software Failures

As mentioned, most previous work has focused on the use of formal methods to support the design of space-related software. In general terms, this approach relies upon the following semantic inconsistency:

$$\text{System, Environment, Requirements} \models \text{false} \qquad (1)$$

In other words, we might wish to establish that a particular model of the system and the environment necessarily involve a violation of safety or liveness properties. This is the traditional role of model checking. These tools will provide a trace of system states and properties that violate particular theorems. This approach can be extremely frustrating. The identification of a semantic inconsistency may provide analysts with limited insights to guide their search for a system and an environment such that the requirements hold. This is not the only way in which formal methods might be used. For example, the following semantic entailment can be used in theorem proving to establish that a system and its environment satisfy a set of requirements:

$$\text{System, Environment} \models \text{Requirements} \qquad (2)$$

In other words, a set of theorems can be shown to hold for a given model of a system operating in a particular environment. These theorems, typically, represent the safety and liveness properties that we might like to hold for our application. This framework is a simplification of the high-level approach to environmental specifications being proposed by Michael Jackson and Pamela Zave [8]. For instance, they have recently proposed the following formalization of 'Adequacy' where e and s represent environment and system models respectively. Environment models include information about the World and any Requirements. System models include information about Machines and Programs:

$$\forall\, e\ s\ .\ \text{World} \wedge \text{Machine} \wedge \text{Program} \Rightarrow \text{Requirements} \qquad (3)$$

In design, these approaches have been used to demonstrate that particular theorems continue to hold, as system models, in other words programs and machines, are iteratively refined towards implementation. We can also use these technologies in a completely different way. For example, after an accident we might like to verify that we have understood the manner in which a failure occurred. For example, one hypothesis about the failure of the Mars Polar Lander mission was that it met a localized meteorological anomaly, such as areas of low pressure, during the parachute descent to the planet surface. In such a situation we might therefore wish to prove

that there exists a revised world model, one in which there are localized low pressure regions, with a machine and program that implies the requirements do not hold:

$$\exists e' \ s \ . \ World \wedge Machine \wedge Program \Rightarrow \neg \ Requirements \qquad (4)$$

Equally, an investigation might focus on potential misunderstandings about the manner in which a program will execute on a particular machine. For example, a software requirement of the Mars Polar Lander was that thrust should be cut to the engines if a signal was generated from the Hall effect sensors on each of the legs and the Doppler radar system detected that the planet surface was in range. However, the programmers failed to account for a global variable that retained a spurious signal that was retained once the legs initially deployed from the body of the Lander. In terms of formula (4) these insights would force us to revise our ideas about how a Program within the system, s, might perform in a particular environment. The key point here is that we can use theorem proving and model checking to demonstrate that changes in our environment or system models will lead to the violation of safety and liveness properties. If we cannot construct such a proof then we need to search for an alternate explanation of the reasons why an accident occurred.

This approach to formal accident verification can yield some interesting surprises. For example, the system and environment models are often correct. In space missions, considerable time and skill is devoted to understanding these issues. The need to understand gravitational influences is well known. Similarly, the bespoke nature of many space missions leads to a detailed understanding of these machines. Mishaps often occur because the safety and liveness requirements are not well understood. For example, the Polar Lander had a software sequence that was to be executed if it remained on the planet surface for 24 hours without receiving a command. The purpose of this software was to start testing alternate communications facilities. However, the Lander was placed into a 'sleep mode' to conserve battery resources with an interval of less than 24 hours. Software reset the timer back to 24 hours each time the Lander awoke and hence the alternate communications configuration was never used. In this example, the model of the world, the machine and the program would satisfy the individual requirements for the backup communications and for the sleep mode. However, the models do not imply the requirement for the backup communications to work in the presence of the sleep mode. This illustrates some of the complexities associated with a formal approach to accident verification by providing an example of the problems associated with the development of complete requirements. The development of a formal proof to identify the potential problem before launch is technically feasible. However, the real challenge is to identify those requirements that are necessary to ensure mission success. Unless we can first do this, there is little likelihood that we will identify the corresponding theorems.

There are few examples of this alternate use of formal methods as a tool to assist accident investigation. Ladkin and Loer have extended theorem-proving mechanisms as part of their Why-Because Analysis technique [12]. This is deliberately intended

to support accident investigation. There are other notable examples. Zuojun Shen [22] has used the Murϕ procedure in Figure 1 to model the Entry, Descent and Landing phase of the Mars Polar Lander. The model checker was used to search for sequences of states that led to the violation of a Murϕ invariant. This stated that the PWM thrust should always be on above a certain altitude. Although Shen's work illustrates the feasibility of the approach, many unresolved questions remain to be addressed.

```
Procedure EDL_DESCENT
        (freeD_uncnty:FREESECENT_ACC_UNCNTY;supon_uncnty:SUPON_ACC_UNCNTY;
        subon_uncnty:SUBON_ACC_UNCNTY;
        subon_hshelloff_uncnty:SUPONHSELLOFF_ACC_UNCNTY;
        SupPyroSwitchHealth: boolean; AccelerameterHEalth: boolean;
        SubPyroSwitchHealth: boolean; AltimeterEalth: boolean);--: EDLstate;
Var ENTRY_OK,state2_OK,state3_OK,state4_OK: boolean;

Begin
        ENTRY_OK :=false; state2_OK:=false; state3_OK:=false; state4_OK:=false;
        if s = ENTRY then
                SupDply(SupPyroSwitchHealth,AccelerameterHEalth,freeD_uncnty);
                ENTRY_OK :=true;
        End;
        if s = state2 & ENTRY_OK then
                SupSepr(supon_uncnty);
                state2_OK:= true;
        End;
        if s = state3 & ENTRY_OK & state2_OK then
                SubDply(freeD_uncnty);
                state3_OK:= true;
        End;
        if s = state4 & ENTRY_OK & state2_OK & state3_OK then
                HeatshellOff(subon_uncnty);
                state4_OK:= true;
        End;
        if s = state5 & ENTRY_OK & state2_OK & state3_OK & state4_OK then
              SubSepr(SubPyroSwitchHealth,AltimeterEalth,subon_hshelloff_uncnty);
        End;
End;
```

Fig. 1. Excerpt from Shen's Model of the Mars Polar Lander Mishap [22]

Traditional Investigation and Identifying Theorems…

The most obvious limitation of formal methods in accident investigation is that the benefits may not outweigh any associated costs. Typically, the budgets available to accident investigation teams are a tiny fraction of those devoted to the development of space missions. Added to this, there are usually tight deadlines by which a report has to be presented to the commissioning authorities. These deadlines are dictated by future launch windows. A number of factors might mitigate these costs. For example, the use of technology such as Murϕ can greatly assist the general application of formal methods both in design and accident verification. By extension, if mathematical specification techniques were more widely used in the development of space systems then this would drastically reduce the costs associated with accident modeling. In other words, we might already have the program, machine and environmental models identified in formula (4).

There are further problems. The application of formal methods would seem to require that we already have some idea about the potential failure mode for the space system. If an existing mathematical model of a program, machine and environment can be shown to violate safety or liveness requirements then the mission should not have gone ahead. In practice many missions, including the Mars Climate Orbiter, have been launched with known bugs in their software. The meta-level point is, however, that we cannot simply set a model checker loose on a system and environmental description with the hope that it will identify a sequence of events leading to an accident. The formalization process necessarily involves a number of complex decisions about the scope of any models and these circumscribe the range of possible causal hypotheses. This problem is even more acute for theorem proving where we must identify the particular safety and liveness properties that are to be disproved. These theorems represent a significant commitment towards the putative causes of an accident. Equally, however, the process of formalization can force developers to ask questions about requirements that might not previously have been asked. This is especially important in the early stages of development before requirements can become intractable in the mass of detail that is associated with an eventual implementation. Unfortunately, the introduction of 'band-aid' software implies that these initial requirements will be subject to constant revision. We are, therefore, faced with a complex situation in which formalization can help both to uncover problems that were not anticipated and to reinforce existing prejudices by modeling those aspects that are already well understood.

It can be argued that a formal model of the symptoms of an accident might be used to support a form of backwards reasoning from the observed failed state. Such models help to narrow the search space of possible causes. However, further problems arise from what has been termed 'causal asymmetries' [10]. If we know that an event has occurred then we can predict its effects with a reasonably degree of confidence. However, if all we know are the consequences of an earlier event then we typically have a far worse ability to predict the causes of those effects. By analogy, if we know a program and its inputs we can reason about the likely outputs. However, if we have a program and its outputs it can be far harder to reason about the combinations of input values that led to the observed results.

The previous caveats undermine some of Shen's achievements in his application of Murϕ to the Mars Polar Lander case study. He already knew what to include in his finite state model because he was working from the Casani report into the mission failure. In general, investigations into space mission failure are not so fortunate. It is worth considering the investigatory processes that did reveal the possible software failures in this mission. The failure mode in the PWM engine code was not found by the application of the Murϕ model checker. Lockhead Martin engineers identified the bug during a test run on a second Lander that was intended for a future mission. An engineer pushed a button to indicate a touchdown too early in the test. He released the button when he realized his error and "was surprised when thrust termination occurred prematurely" [19]. This prompted a more formal failure analysis that uncovered the software problem. Similarly, the bug in the Polar Lander's uplink

command string was not found during the initial code design walkthrough. The investigators argued that one reason for this was that logic flow diagrams were not used; “it is difficult to find logic errors by walking through the code without logic flow diagrams to help the process” [19]. The uplink design error was discovered after a fault-tree analysis led to the examination of the code and the preparation of code descriptions for reviews by outside reviewers. Such observations make it important to be careful in the claims that are made for the formal analysis of accidents. They can be used to add confidence in any analysis but, at present, it seems too optimistic to argue that they will automatically uncover failure modes. It seems likely that the use of mathematical reasoning will continue to depend upon insights provided by more traditional forms of software forensics [9].

Material Implication Does Not Represent Causation

The previous section focused on some of the practical limitations to the formal verification of accident models. There are also a number of theoretical problems [11]. For example, many people would interpret formula (4) as representing a causal relationship. Changes in our environmental model can be used to explain why an accident occurred. Unfortunately, material implication cannot easily be used to represent and reason about the causes of an adverse event. Several paradoxes, including circular arguments, can confuse the unwary. The impact of these paradoxes and other features of material implication should not be underestimated. For instance, we can introduce an arbitrary true antecedent to implications that may convince non-mathematicians of causal relationships even though there is no direct relevance with the antecedent. ‘If NASA’s Faster, Better Cheaper programme reduced funds for the Mars Surveyor projects then software failures led to the loss of the Polar Lander’.

A number of logicians, philosophers and linguists have recognised the limitations of strict implication and have responded by constructing alternative logics, which avoid the problems of classical logic, or by analysing the ways in which people construct implicational statements using material conditions. Grice [5] and Jackson [7] have exploited this latter approach. They argue that material implication remains a valid form of argument for *indicative* conditionals. In particular, Grice and Jackson observe that most people use arguments to communicate information in the most ‘cost effective’ means possible. They are anxious to avoid the costly repair actions that are necessary whenever misunderstandings occur. One consequence of this is that people will not assert weaker forms of a proposition when they can assert a strong form. In particular, speakers do not say 'If *P*, then *Q*' when they know that *P* is false. It is simpler and more informative to say 'not *P*'. Grice and Jackson’s analysis is important because it can be used to avoid some of the problems that arise from material implication between two arbitrary false statements. Recall that material implication would allow a statement of the form 'If snow is black, then grass is red‘ to be true. Grice and Jackson argue that people do not reject such statements because they believe them to be ‘false’. Instead, they argue that our reservations stem from

the impression that such arguments would misleadingly suggest that we are unsure about the colour of snow.

Lewis [see Lewis and Langford, 13] goes beyond the material implication of classical logic to develop the notion of strict implication. This is based upon the idea that a proposition *strictly implies* all others, which are true, in all possible circumstances where it is true. The semantics for this form of strict implication is based around that of modal logics. Hence, we have that *A->> B* is true at world *w* if and only if for all *w'* such that *w'* is accessible to *w*, either *A* fails in *w'* or *B* obtains there. However, the Lewis semantics for strict implication still permit an antecedent that is irrelevant to the consequent. Logicians have responded by developing what are known as relevance logics. One approach builds on a notion of 'relevant' proof [1]. This requires that premises and conclusion must share a variable in valid conditionals. This requirement can help to ensure that the antecedent and consequent refer to the same object in an assertion. Alternatively, the proof theory of relevance logics can require that conclusions can be directly derived from a premise without the introduction of arbitrary antecedents and consequents. This is intended to ensure that any premises really are used to obtain a valid conclusion.

Further problems also arise because the material implication of classical logic cannot convey different and varied interpretations of causal information. For example, mishap investigators often distinguish between necessary and sufficient causes. A necessary cause is often identified using counter-factual arguments of the form 'the mishap would not have occurred if this cause(s) had not also occurred'. A sufficient cause can be distinguished by arguments of the form 'the mishap could have occurred if this cause(s) had taken place irrespective of any other of the other circumstances surrounding the incident'. Similarly, many causal arguments are constructed using a form of subjunctive conditional that is not characterized by material implication. In particular, counterfactual conditionals rely upon an antecedent, which represents a past tense subjunctive sentence of the form "If X *had been the case* …then Y would have happened. These sentences are known as counterfactuals because there is an assumption that the antecedent is false. In other words that X is known not to have been the case. For example, an investigator might assert that 'If he had been further away, then he would not have been hurt'. There is an implication that he was NOT further away and also that he was, in fact, hurt. Most incident investigation guidelines explicitly recommend that investigators use counterfactual arguments to guide their analysis [11]. The Lewis semantics for strict implication can be used to form counterfactual arguments. However, the interpretation of the accessibility relation between possible worlds still relies on the subjective judgment of domain experts. In other words, disagreements can arise over whether it is plausible that an accident would have been avoided if only a cause had been prevented.

The key meta-level issue here is that many of the logics that are used to support the formal analysis of complex systems have serious limitations if they are to model the causes of incidents and accidents. Instead, we have been forced to rely on modal logics and non-standard proof techniques. The identification of a tractable alternative

to first order classical logic remains a topic of considerable debate amongst the small number of researchers in this area. It also remains the focus of several funding initiatives from the potential end-users of this technology.

Can We Model the System and the Environment?

The opening sections of this paper described how many academic software engineers use space mission failures to warn students about the hazards of programming. Many of these talks omit critical details. For example, they focus on the confusion between imperial and metric units in the Mars Climate Orbiter code. They overlook the ways in which software was used as a 'band aid' for the asymmetrical solar arrays. Similarly, I have attended research talks where software engineers construct elaborate counterfactual arguments of the form 'if only ESA/NASA/ISRO had followed software engineering technique X then the mishap would have been avoided'. Such counterfactuals are by their very nature non-truth functional. We have no accessible world in which the mishap did not occur so we can never really be sure that the software engineering technique X would have prevented the mission failure.

In contrast, I would urge software engineers to watch more natural history programmes on television. These programmes help to show how the animals' environment helps to shape behavior. By analogy, in order to understand the causes of software failure in space missions we need to look beyond the immediate causes of bugs to look at the organizational context that created them. It is extremely fashionable to talk about accidents as the result of 'emergent properties' or unanticipated outcomes from interaction between subsystems. I do not support this view. All of the failures mentioned in this report had precursors; the agencies either had experienced previous similar failures or their own employees and sub-contractors had described potential concerns through incident reporting systems.

It is also important to stress that analytical techniques can be applied to represent and reason about the environment in which bugs occur. For example, Figure 2 represents an Event and Causal Factor (ECF) analysis for the pre-launch phase of the Mars Polar Lander [10]. The US National Transportation Safety Board and the US Department of Energy pioneered ECF for use in accident investigation. Rectangles denote events while ellipses are used to represent those causal factors that make events more likely.

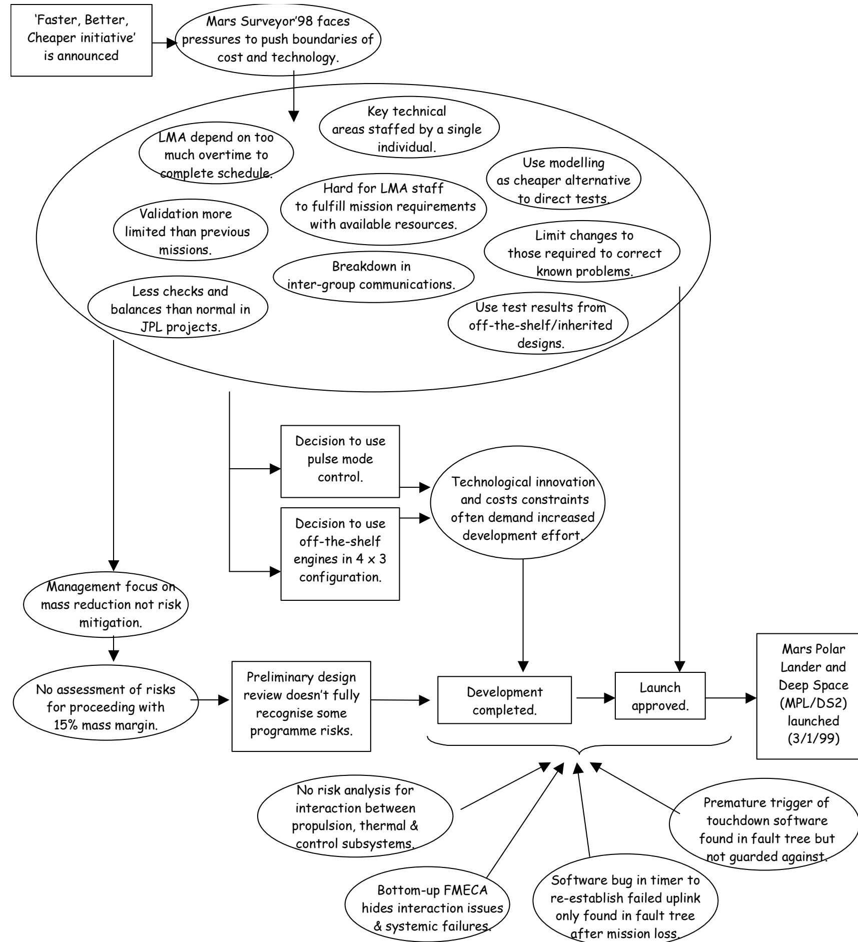

Fig. 2. Events and Causal Factor Overview of the Mars Polar Lander, Pre-Lauch [10]

The 'Faster, Better, Cheaper' initiative placed the entire Surveyor programme under pressure to push the boundaries of cost and technology. This in turn led to a number of contextual factors that helped shape the programming effort. It was hard for contractors to meet the mission requirements with the available resources. As we have seen, opportunities for testing and validation were restricted as tight deadlines prevented access to hardware platforms and costs prevented many forms of destructive testing. Analysis and modeling were proposed as lower cost alternatives and so on. These influences led to the decision to use pulse mode control and a 4 by 3 array of off the shelf engines in preference to previous missions that had used a more gradual form of throttle control. The outcome of these decisions was to increase the complexity of software development to control the platform. At the same time,

management focused on the problems of mass reduction so that the Polar Lander would meet the performance profile of the launch vehicle and cruise resources. This arguably took their attention away from the wider engineering risks created by cost reduction across the programme. The contextual factors at the bottom of Figure 2 show that fault tree analysis revealed the hazard from premature shutdown of the Lander engines, possibly triggered by a software bug. However, this risk was not adequately guarded against.

Figure 2 characterizes the growing pressures on investigators to look beyond the immediate or catalytic events that lead to mission failures. It has been argued by government organizations, by researchers and by a mass of other public bodies that accident and mishap analysis should instead look for root causes [10]. Unfortunately, software engineering has a tendency to focus on the immediate events that trigger particular failures. We remember the code re-use in Ariane 5 or the metric and imperial confusion with the Mars Climate Orbiter or even the uplink timer commands on the Polar Lander. Instead, we should look at the underlying causes. For instance, the Faster, Better Cheaper initiative arguably fostered a culture in which engineers took considerable risks to innovate with new design. These included the asymmetrical solar arrays on the Climate Orbiter and the pulse controlled engines on the Polar Lander. These innovative engineering decisions saved costs but relied on 'band aid' software. Programmers were forced to calculate the de-saturation parameters that would compensate for momentum induced by the innovative solar arrays. Programmers had to develop control software for the pulse times needed by the Polar Lander.

Conclusions and Further Work

The rise of 'systemic' approaches to accident investigation has clear implications for the use of formal methods in mishap analysis. One option is to follow the route taken by many others in the formal methods communities by looking for niche applications. Mathematical reasoning might be confined to the early stages of an investigation where it is important to understand precisely what happened. In this view, techniques such as model checking would provide simple extensions of their more conventional role in software engineering following the model outlined by Shen's use of the Murϕ system. The challenges of this work should not be underestimated. In particular, we must find ways of using the results from theorem proving and model checking to inform the wider analytical techniques, such as ECF analysis, that will retain the primary role in identifying the managerial and organizational root causes of any mishap. This use of formal methods in forensic software engineering raises a host of further technical barriers. Space-related software continues to become more complex as it controls increased functionality and provides a vehicle for highly integrated systems, including satellite arrays.

An alternative future is one in which the scope of formal methods is expanded to reason about the root causes of software-related failures. Such a route follows the

vision of Jackson and Zave where we begin to model many features of the environment that are not traditionally considered within formal areas of software engineering. Again this poses enormous technical challenges. A key question is what might be included within a formal model of a mishap. For interactive systems, such as the Shuttle's General Purpose Computing system, our model may be forced to consider cognitive, perceptual and physiological attributes of the crew. This, in turn, raises profound questions about the abstractions that might support such modeling. There has been work on formal aspects of human computer interaction but the results are limited and can often be disappointing when applied to applications such as the Shuttle or Rovers. Even if formal modeling were expanded in this way, it would still not capture the organizational and managerial issues that are increasingly being identified as the root causes of software failure. The use of epistemic and deontic notations to model such decision-making now forms part of the heritage of formal methods. Studies in the 1980s and 1990s showed how these techniques might be used, for instance to model legislative requirements. Again, however, the results do not seem to scale well and there are considerable problems in developing suitable proof theories. These problems are compounded when one remembers the host of problems in developing discrete mathematics to provide a satisfactory model of causal arguments.

To summarize, this paper has introduced some of the demands that are created by software development for space-related applications. These include the usual suspects that complicate all forms of software engineering. However, the physical properties of space environments create novel problems. For example, data and software updates must often be communicated over vast distances and this creates novel forms of batch processing. High-levels of radiation as well as mass and power limitations also create problems because they typically force programmers to rely on specialist hardware. Additional verification requirements and the limited sales of these processors often imply that they are obsolete in terms of mass-market applications long before they reach the launch pad. Later sections have also described the problems created by 'band aid' software. There is a growing tendency to rely on code to mitigate problems created by engineering decisions that are made elsewhere in the development of a space mission. One consequence of this is that software seems to be playing an increasingly prominent role in space-related mission failures.

The traditional role of formal methods can be expanded beyond design to analyze software failures. Existing models of software development, such as that proposed by Jackson and Zave, can easily be adapted to support this endeavor. Others have used a range of theorem proving and model-checking technology to represent and reason about space-related software failures [10, 22]. However, there are many technical and conceptual challenges that remain to be addressed. In particular, software bugs often form part of more complex problems that permeate through many different aspects of the engineering of space missions. The technical challenges also include basic issues with the representation of causal arguments given the limitations of classical material implication. The conceptual issues relate to the scope of the

modeling activity. Do we focus narrowly on the behavior of a machine and its program? Or do we consider the managerial and organization precursors that are the root causes of software failure? Until these issues are resolved we remain even less equipped to identify the causes of software failure than we are to support the development of space related systems.

References

[1] A.R. Anderson and N.D. Belnap, Entailment: The Logic of Relevance and Necessity, Princeton, Princeton University Press, Volume I, 1975.

[2] J.Blum, Intelsat Loses Use of Satellite: Spacecraft Failure Could Jeopardize Sale of Company, Washington Post, Tuesday, January 18, 2005; Page E01.

[3] J. Crow and B. L. Di Vito. Formalizing space shuttle software requirements. In Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice, pages 40-48, January 1996.

[4] L. Devauchelle, PICGAL: Process Improvement Experiment of a Code Generator to the ARIANE Launcher, ESSI Project 21 710, Final Report, Aerospatiale, November 1997. http://www.esi.es/VASIE/Reports/All/21710/Report/21710.pdf

[5] H.P. Grice, Studies in the Way of Words. Harvard University Press, Cambridge MA, 1989.

[6] R. Hillman, M. Conrad, P. Layton, C. Thibodeau, G.M. Swift and F. Irom, Space Processor Radiation Mitigation and Validation Techniques for an 1800 MIPS Processor Board, Maxwell Technologies and Jet Propulsion Laboratory, California Institute of Technology, 2003. http://parts.jpl.nasa.gov/docs/radecs03_swift.pdf

[7] F. Jackson, On Assertion and Indicative Conditionals. Philosophical Review, (88):565-589, 1979.

[8] M. Jackson and P. Zave, Deriving Specifications from Requirements: An Example, Proceedings of the 17th International Conference on Software Engineering, pages 15-24, ACM Press, 1995.

[9] C.W. Johnson, Forensic Software Engineering: Are Software Failures Symptomatic of Systemic Problems? Safety Science (40)9:835-847, 2002.

[10] C.W. Johnson, A Handbook of Accident and Incident Reporting, Glasgow University Press, Glasgow, 2003. http://www.dcs.gla.ac.uk/~johnson/book

[11] C.W. Johnson and C.M. Holloway, A Survey of Causation in Mishap Logics, Reliability Engineering and Systems Safety, (80)3:271-291, 2003.

[12] P. Ladkin and K. Loer, Why-Because Analysis: Formal Reasoning About Incidents, RVS-Bk-98-01, Technischen Fakultät der Universität, Bielefeld, Germany, 1988.

[13] C.I. Lewis and C.H. Langford, Symbolic Logic, The Century Co. New York and London, 1932.

[14] J.L. Lyons. Report of the inquiry board into the failure of Flight 501 of the Ariane 5 rocket. Technical report, European Space Agency, Paris, France, July 1996.

[15] NASA, Expedition 10: Paving the Road for the Return to Flight, International Space Station, Science Operations, Oct. 2004. http://www.scipoc.msfc.nasa.gov/expedition10.html

[16] NASA. Mars Climate Orbiter: Mishap Investigation Board, Phase I Report. Technical report, Mars Climate Orbiter, Mishap Investigation Board, NASA Headquarters, Washington DC, USA, 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[17] NASA, Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Report NASA-GB-002-95, NASA Office of Safety and Mission Assurance, Washington DC, 1995. http://eis.jpl.nasa.gov/quality/Formal_Methods

[18] NASA/ESA, SOHO Mission Interruption Joint NASA/ESA Investigation Board Final Report, 1998. http://umbra.nascom.nasa.gov/soho/SOHO_final_report.html

[19] NASA/JPL. Report on the loss of the Mars Polar Lander and Deep Space 2 Missions (The 'Casani' Report). JPL D-18709, NASA/Jet Propulsion Laboratory, 2000.

[20] NASA/JPL, Sol 243-262: Spirit Back to Normal Operations, Mars Exploration Rover Mission, NASA/Jet Propulsion Laboratory, California Institute of Technology, 29 September 2004, http://marsrover.nasa.gov/mission/status_spiritAll.html#sol243

[21] NASA/JSC EVA Project Office, EVA Equipment Board (EEB) Minutes of Meeting September 19, 2001, http://www.spaceref.ca/news/viewsr.html?pid=3821

[22] Z. Shen, Model Checking for the MPL Entry and Descent Sequence, Technical Report, Department of Aerospace Engineering, Iowa State University, December 2001, http://www.public.iastate.edu/~zjshen/ProjectReport.pdf