Questioning the Role of Requirements Engineering in the Causes of Safety-Critical Software Failures

C. W. Johnson[†], C. M. Holloway*

[†] Dept. of Computing Science, University of Glasgow, Glasgow, G12 9QQ, johnson@dcs.gla.ac.uk
*NASA Langley Research Center, 100 NASA Road, Hampton VA 23681-2199, U.S.A, c.m.holloway@nasa.gov

Keywords: software failures, requirements engineering, accident analysis, fallacious reasoning

Abstract

Many software failures stem from inadequate requirements engineering. This view has been supported both by detailed accident investigations and by a number of empirical studies; however, such investigations can be misleading. It is often difficult to distinguish between failures in requirements engineering and problems elsewhere in the software development lifecycle. Further pitfalls arise from the assumption that inadequate requirements engineering is a cause of all software related accidents for which the system fails to meet its requirements. This paper identifies some of the problems that have arisen from an undue focus on the role of requirements engineering in the causes of major accidents. The intention is to provoke further debate within the emerging field of forensic software engineering.

1 Introduction

The last twenty years have seen a move away from the ‘perfective approach’ in accident investigation [1]. Rather than blaming the operators who directly control complex applications processes, there has been an increasing tendency to identify the underlying latent and distal causes of adverse events, such as poor safety management [2].

There have been similar changes in forensic software engineering. Investigators have looked beyond particular bugs to identify problems in the wider procurement and management of large IT projects. In particular, there has been a growing emphasis on requirements engineering in the causes of adverse events. Leveson has argued that “most software reliability models define failures in terms of deviations from the software requirements specification; most accidents involving software are due to errors in the software requirements specification” [3].

Similarly, Ladkin’s analysis of the loss of Ariane 5 focuses on requirements errors rather than programming failures: “the program was written against Ariane 4 requirements; these requirements were not transferred to the Ariane 5 requirements spec; the Ariane 5 requirements therefore did not state the range requirement; the (implicit in Ariane 5) range requirement was in conflict with the behavior of Ariane 5 (as in fact explicated in other Ariane 5 requirements); requirements came up against behavior and the rocket was destroyed. (It is not surprising that it was a requirements error - over 90% of safety-critical systems failures are requirements errors, according to a JPL study that has become folklore, as well as Knight-Leveson, I believe.)” [4].

The following, partial list provides examples of the types of problems that can arise during these early stages of development:

1. *lack of stakeholder involvement.* The end-users who arguably know most about day to day operation may not be sufficiently consulted in the early stages of development.
2. *incorrect environmental assumptions.* Neumann's collection of computer related risks contains numerous examples of variables that have fallen above or below their anticipated ranges during 'normal' operation [5].
3. *communications failures within development teams.* Software engineers must often rely upon information provided by domain experts. Problems arise when these specialists must communicate technical expertise to people from other disciplines [6].
4. *inadequate conflict management.* Different stakeholders can hold radically different views about the purpose and priorities of application software. Such disagreements can result in inconsistent or missing requirements if they are not addressed.
5. *lack of contextual detail.* Requirements cannot simply be gathered by conducting interviews or by analyzing existing documentation. Observational techniques have been used to provide first-hand insights into the potential operational environment of complex, safety-critical systems.

Statistical evidence has been gathered to demonstrate the importance of requirements capture in the development of safety-critical systems. For example, the UK Health and Safety Executive’s ‘Out of Control’ project conducted a detailed review of the causes of software failures in process control applications. Requirements issues accounted for 40% of the incidents, hardware failures for 26%, software bugs 11%, maintenance issues 6% and ‘system use’ around 17% [7].

Also, Vinter [8] analysed more than 1,000 bug reports produced by seven major embedded real-time systems projects. He found results that are broadly similar to those obtained by the UK HSE: 24% of the bug reports stemmed from requirements issues, functionality 25%, structure 21%, data 10%, implementation 5%, integration 5%, architecture 1%, testing 7%, other 5%. Within those bug reports that were associated with requirements problems, Vinter argued that 48% could be classified as 'misunderstandings'. Typically, disagreement existed over the precise interpretation of a particular requirement. 19% of the bug reports that stemmed from requirements problems related to missing constraints. 27% related to requirements that had been changed. A further 6% were classified as 'other' issues. These statistical studies have inspired some researchers to advocate a new vision of safety-related requirements engineering. One such vision would integrate pure safety requirements, safety-significant requirements, system safety requirements, and safety constraints in requirements repositories of requirements specifications [9].

Nuseibeh and Easterbrook have argued that "the primary measure of success of a software system is the degree to which it meets the purpose for which it was intended. Broadly speaking, *software systems requirements engineering* (RE) is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation" [10]. These observations correctly emphasize the importance of requirements engineering for the development of complex systems.

Problems arise when such arguments become distorted by a form of causal asymmetry. Requirements engineering helps to ensure that the software meets the primary measure of success. However, this does not necessarily imply that software failures are *all* caused by a failure in requirements engineering. Unfortunately, many previous academic studies and accident investigations seem to suffer from just this form of hindsight bias [11]. To paraphrase: there must have been a failure in requirements capture because engineers would have fixed the problem if they had known about it.

There are further concerns about the statistical basis for the studies that have emphasized the importance of requirements engineering. For instance, Vinter's study was based on previous work by Beizer [12]. This earlier work had looked at more than 16,000 bug reports in major software projects funded by the US government. In contrast to Vinter, Beizer only classified 8% of the problems as stemming from requirements. It seems unlikely that the US government is so much better at requirements analysis than the commercial project teams investigated by Vinter. Instead, the difference in results can be explained in terms of the classification systems that were used in these studies. Beizer relied on a classification system with several dozen terms while Vinter relied on only nine top level categories. The key point here is that a superficial analysis of software engineering failures will often focus on requirements failure, because of the persuasive nature of the hindsight bias mentioned above. In contrast, more detailed studies in forensic engineering tend to focus on a host of organisational and technical issues that are less easily classified.

2. Case Study

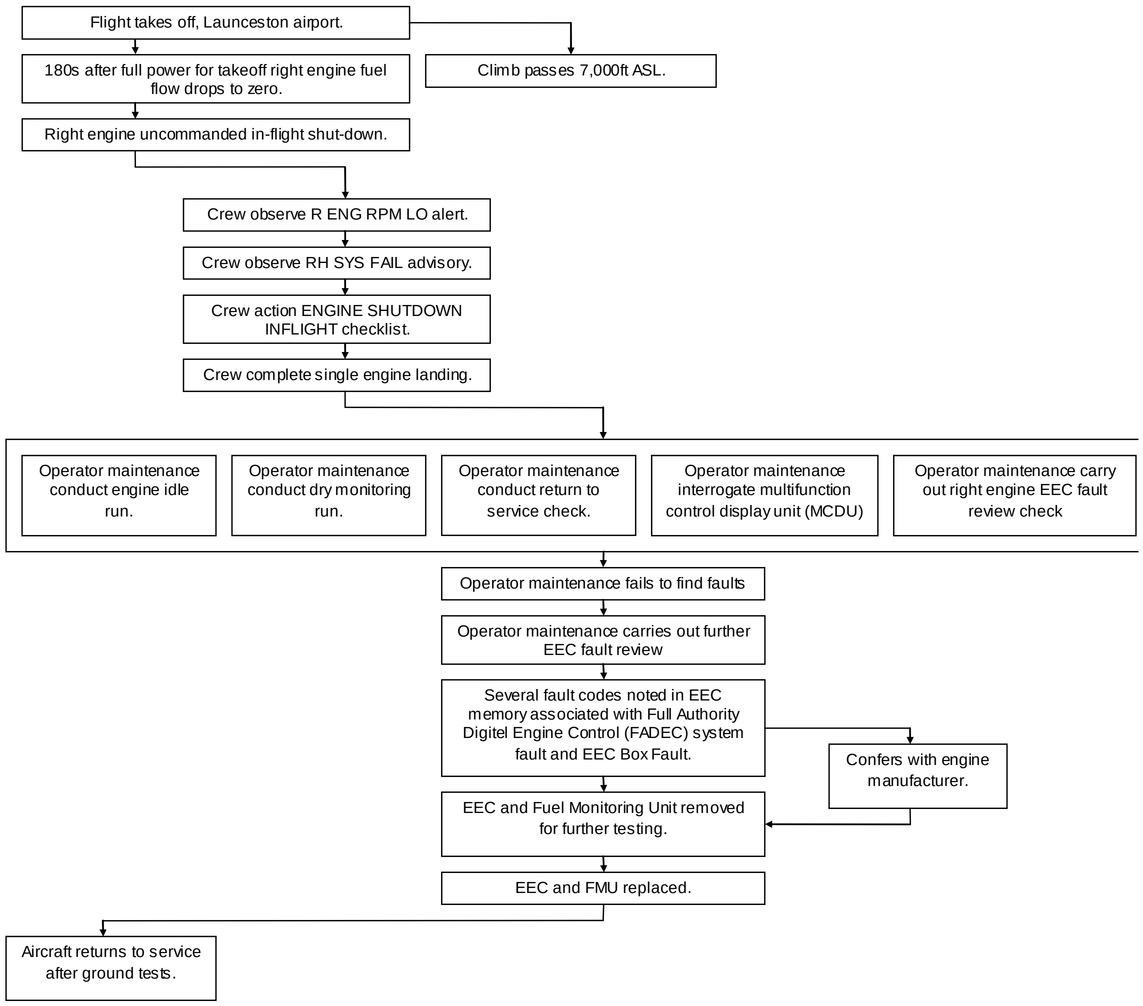
Arguments about the role of requirements engineering in the failure of safety-critical systems can be illustrated by a recent incident involving Boeing Electronic Engine Controllers (EEC) [13]. These units control engine start sequencing, power requirements, operating temperature, turbine speeds, fuel flow, engine monitoring, and automatic relight, among other functions. They also provide fault detection using EEPROMs and log error codes until they are intentionally cleared during maintenance.

The particular incident in which we are interested was triggered when the fuel flow to the right engine dropped to zero approximately 3.5 minutes after full power was applied for take-off during a commercial flight. This resulted in an in-flight shutdown for the engine. The crew received no warning before the failure, but they were able to successfully land with only one engine. Figure 1 provides a simple Events and Causal Factors diagram for the incident. Rectangles denote individual events, ellipses represent causal factors that make those events more likely.

Subsequent investigations found that a failure in the right engine EEC removed electrical power from an engine fuel-control metering valve. This valve was spring loaded to return to the closed position when power was removed. The EEC removed the electrical signal from the fuel valve, because it had detected failures on both of its redundant channels. Channel A suffered from a bit-flip error within the memory section of the input/output microprocessor. Unused memory sections should have been initialized using a positive charge to represent binary 1. However, some areas became negatively charge and were interpreted as binary zero, resulting in checksum failures.

Further vibratory testing of the EEC indicated fracturing of solder joints at five resistors on the analog interface module circuit board of Channel B. It was discovered that the EECs in other fleets had also suffered from these problems. At first sight, this incident can be interpreted as a requirements problem similar to those identified in the second item of the list that opened this paper. The programmers made incorrect assumptions about the reliability of the operating environment for their software. The decision to initialize memory section of the input/output microprocessor using positive charge binary 1, left the system susceptible to in-flight failures if areas became negatively charged by individual bit-flip errors. This was considered to be a more likely failure mode than a bit-flip from negative 0 to positive 1. In consequence, two months after the incident, the engine manufacturers issued a Service Bulletin which gave instructions for a software modification of the processor communication's modules to change the fill pattern of the unused areas of the EEPROM memory from hexadecimal binary 1s to binary 0s, thereby reducing the possibility of checksum failures.

One problem with this particular analysis is the difficulty of determining whether particular problems stem from



Flight takes off, Launceston airport.
Climb passes 7,000ft ASL.
180s after full power for takeoff right engine fuel flow drops to zero.
Right engine uncommanded in-flight shut-down.
Crew observe R ENG RPM LO alert.
Crew observe RH SYS FAIL advisory.
Crew action ENGINE SHUTDOWN INFLIGHT checklist.
Crew complete single engine landing.
Operator maintenance conduct engine idle run.
Operator maintenance conduct dry monitoring run.
Operator maintenance conduct return to service check.
Operator maintenance interrogate multifunction control display unit (MCDU)
Operator maintenance carry out right engine EEC fault review check
Operator maintenance fails to find faults
Operator maintenance carries out further EEC fault review
Several fault codes noted in EEC memory associated with Full Authority Digitel Engine Control (FADEC) system fault and EEC Box Fault.
Confers with engine manufacturer.
EEC and Fuel Monitoring Unit removed for further testing.
EEC and FMU replaced.
Aircraft returns to service after ground tests.

Figure 1: Overview of the Electronic Engine Controller (EEC) Failure

inadequate requirements engineering, or from other stages of the software lifecycle. For example, it might be argued that the initialization conditions for the EEC input/output microprocessor should only be considered during detailed specification and design, and not during requirements development Equally, however, it might be argued that the use of positive 1 encoding violated an earlier requirement to ensure that the system was resilient to bit-flip errors. Hence, the problem can be traced back to the verification and testing of a design against high-level requirements. We shall return to this issue in the discussion at the end of the paper.

Hindsight bias occurs when software engineers automatically assume that there has been a failure in requirements engineering simply because an accident has occurred. This is dangerous if it obscures deeper engineering problems. For example, 'band aid' software can be applied to fix new requirements rather than address more fundamental engineering issues.

In the accident we are considering, investigators may focus on changes to version 7.0 of the EEC software rather than on problems in the printed circuit boards, which are illustrated in Figure 2. This diagram extends the ECF modeling to identify more detailed causes of the EEC failure.

Our earlier discussion focused on the checksum failures that were associated with Channel A, but did not address the failure of Channel B, which was also necessary in order to cause a failure in the dual redundant EEC for the right engine. The ECF diagram denotes that Channel B may have failed from differential thermal expansion between the printed circuit board and a series of interface resistors. Over time, these failures contributed to 'health-lane degradation' which ultimately resulted in Channel B being shut down. As can be seen, the 'health lane degradation' on both Channel A and Channel B were difficult to detect because intermittent failures did not always result in maintenance warnings. The Australian Transport Safety Board went on to describe how

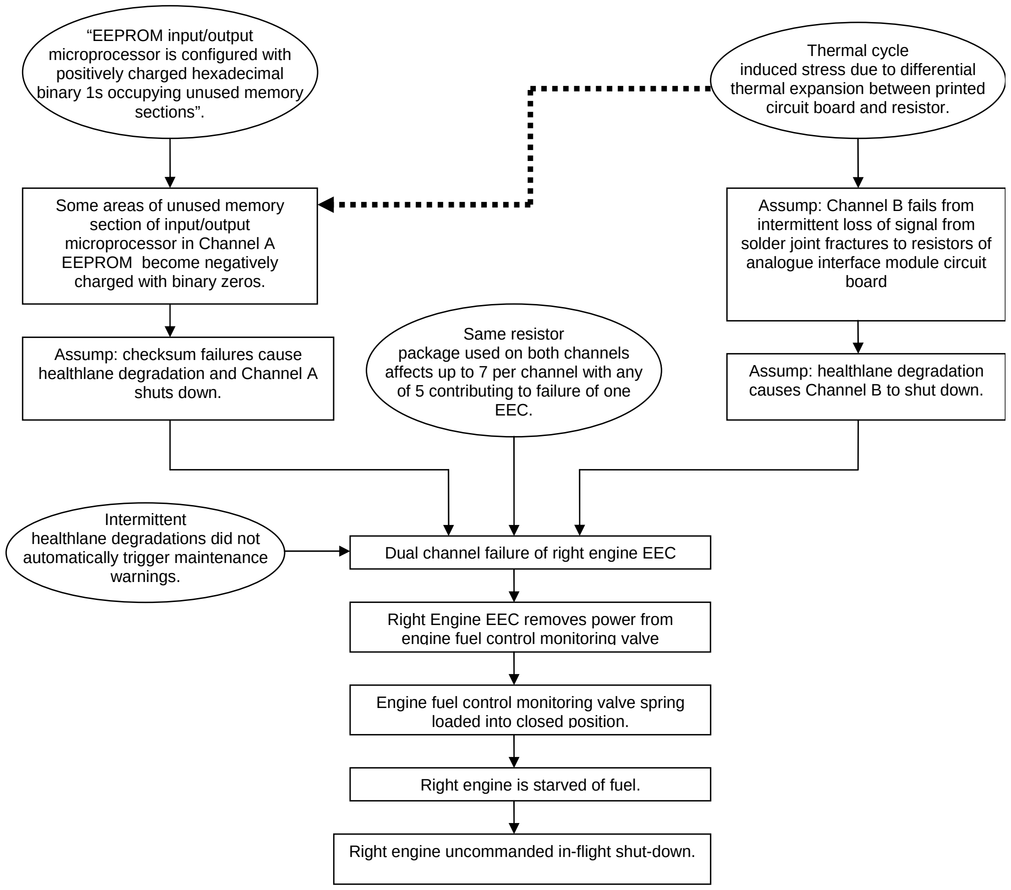

"EEPROM input/output microprocessor is configured with positively charged hexadecimal binary 1s occupying unused memory sections".
Thermal cycle induced stress due to differential thermal expansion between printed circuit board and resistor.
Some areas of unused memory section of input/output microprocessor in Channel A EEPROM become negatively charged with binary zeros.
Assump: Channel B fails from intermittent loss of signal from solder joint fractures to resistors of analogue interface module circuit board
Assump: checksum failures cause healthlane degradation and Channel A shuts down.
Same resistor package used on both channels affects up to 7 per channel with any of 5 contributing to failure of one EEC.
Assump: healthlane degradation causes Channel B to shut down.
Intermittent healthlane degradations did not automatically trigger maintenance warnings.
Dual channel failure of right engine EEC
Right Engine EEC removes power from engine fuel control monitoring valve
Engine fuel control monitoring valve spring loaded into closed position.
Right engine is starved of fuel.
Right engine uncommanded in-flight shut-down.

Figure 2: Detailed Analysis of the Electronic Engine Controller (EEC) Failure

"the engine manufacturer further advised that they will be incorporating a software upgrade of the EEC to version 7.0, which will include an improvement to remove the possibility for certain intermittent failures to trigger a 'health lane' degradation without triggering the corresponding maintenance message".

Figure 2 illustrates further design issues that were not so prominently discussed in the official incident report and which might also easily be overlooked by a precipitate focus on requirements engineering issues. Both of the redundant Channels were designed using similar resistor packs. In consequence, both were vulnerable to the same thermal cycles. This more detailed analysis points to more complex causes in the incident than the initial failure to identify appropriate initialization conditions for the EEPROM input/output microprocessor software. It also suggests continuing vulnerabilities from a lack of diversity even after the proposed software upgrades.

Figure 3 provides an overview of a second incident, *involving the same aircraft*. Approximately three weeks after the right engine EEC failure, there was an in-flight loss of power to the left engine. The causes appear to have been very similar.

The associated EEC was also running version 6.1 of the software. Subsequent inspections revealed fractures to the solder joints on the resistors of both redundant channels. Maintenance teams were able to identify an EEC fault code in the multifunction control display unit (MCDU) memory.

As in the previous incident, the crew landed the aircraft without any injury. Six days after this second incident, there was a further in-flight loss of power reported from a US aircrew. Again, there were fractures in the solder joints across resistors for both of the channels.

This case study initially focused on requirements problems associated with the initialisation of the EEC's EEPROM input/output microprocessor. Subsequently we identified software problems in logging intermittent failures during health lane monitoring. Although these problems can be linked back to the requirements analysis for the EEC application, Figures 2 and 3 show that they must be placed in

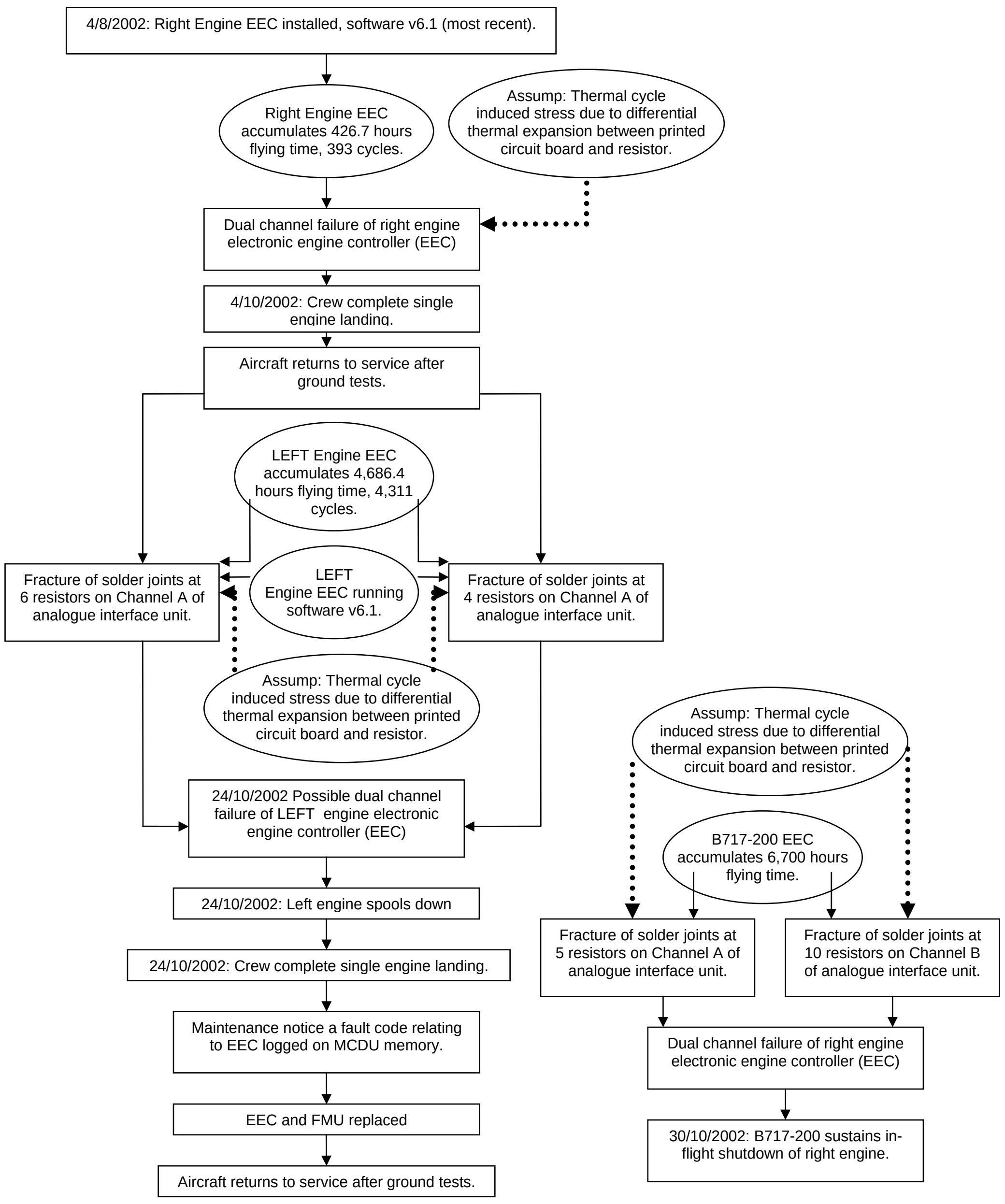
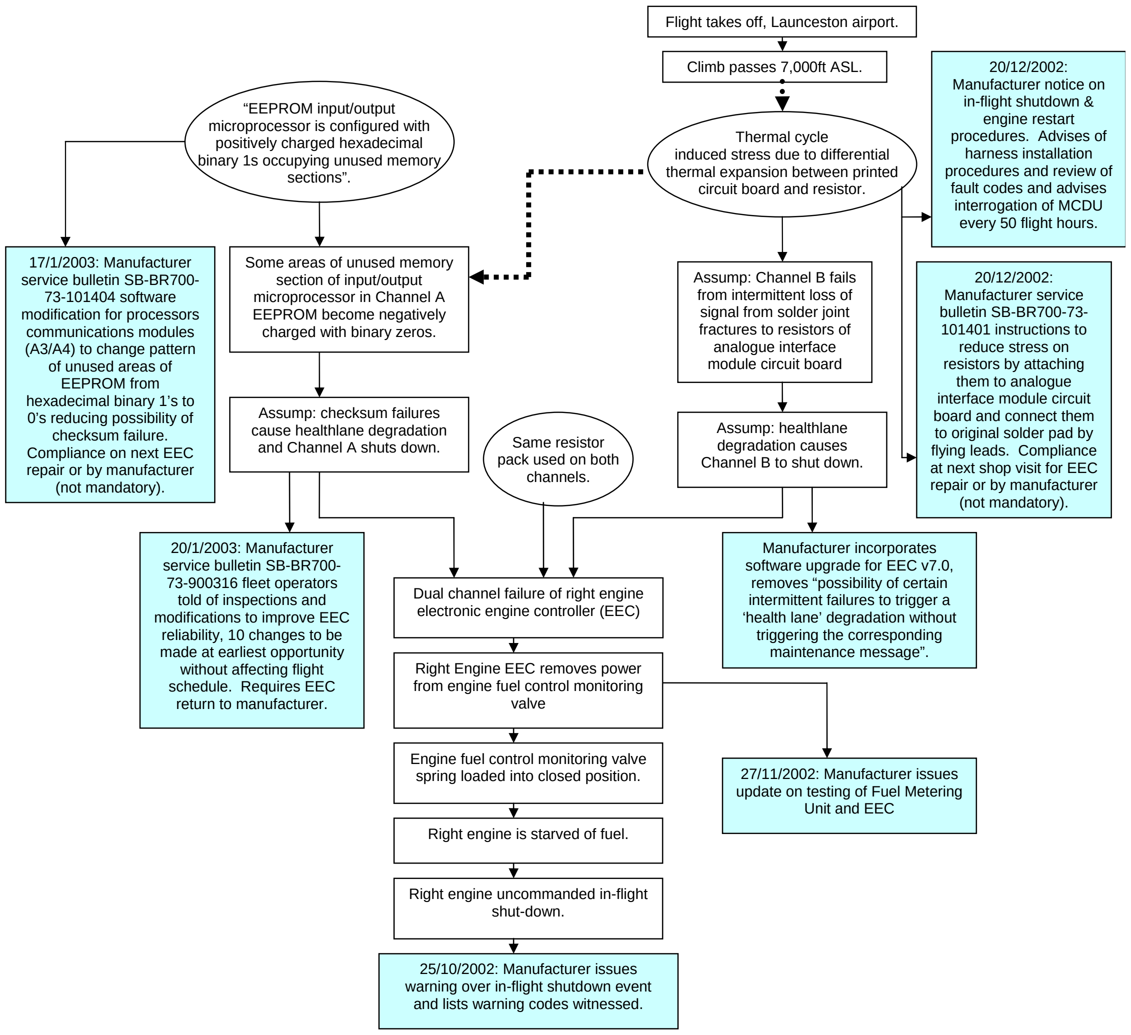

4/8/2002: Right Engine EEC installed, software v6.1 (most recent).
Right Engine EEC accumulates 426.7 hours flying time, 393 cycles.
Assump: Thermal cycle induced stress due to differential thermal expansion between printed circuit board and resistor.
Dual channel failure of right engine electronic engine controller (EEC)
4/10/2002: Crew complete single engine landing.
Aircraft returns to service after ground tests.
LEFT Engine EEC accumulates 4,686.4 hours flying time, 4,311 cycles.
Fracture of solder joints at 6 resistors on Channel A of analogue interface unit.
LEFT Engine EEC running software v6.1.
Fracture of solder joints at 4 resistors on Channel A of analogue interface unit.
Assump: Thermal cycle induced stress due to differential thermal expansion between printed circuit board and resistor.
24/10/2002 Possible dual channel failure of LEFT engine electronic engine controller (EEC)
24/10/2002: Left engine spools down
24/10/2002: Crew complete single engine landing.
Maintenance notice a fault code relating to EEC logged on MCDU memory.
EEC and FMU replaced
Aircraft returns to service after ground tests.
Assump: Thermal cycle induced stress due to differential thermal expansion between printed circuit board and resistor.
B717-200 EEC accumulates 6,700 hours flying time.
Fracture of solder joints at 5 resistors on Channel A of analogue interface unit.
Fracture of solder joints at 10 resistors on Channel B of analogue interface unit.
Dual channel failure of right engine electronic engine controller (EEC)
30/10/2002: B717-200 sustains in-flight shutdown of right engine.

Figure 3: Similar Incidents of Electronic Engine Controller

the context of several wider engineering issues. In particular, any focus on software requirements issues must be balanced against a lack of diversity in the resistor arrays of both channels and the configuration of the PCBs that made then susceptible to thermal cycles.

The key point here is not to deny that there were requirements problems in this case study. In contrast, the intention is to warn against the myopia that occasionally affects software engineering accounts of major failures [3]. For example, there are well-developed research communities in the field of requirements engineering. This increases the likelihood that research projects will focus on these aspects of adverse events [11]. However, in this case study, equal attention should be paid to the wider engineering issues mention above. It is also important to consider the role of software engineering in monitoring the error codes produced by the EEC and logged in MCDU memory. It seems remarkable that several almost identical failures were logged within weeks of the first EEC failure. This suggests an increased awareness of the possible failure modes given successive updates from the manufacturer. There have been a number of recent research and development initiatives to support pattern recognition and self-diagnosis of potential failure modes from on-board systems [14]. However, this work is largely driven by aviation specialists and cannot easily be compared to the mass of recent projects in the more general field of requirements engineering.

Figure 4 reinforces many of the points made in previous paragraphs by annotating an ECF with specific actions taken by the manufacturer in response to this incident. Shaded rectangles denote interventions that address diverse causes ranging from the initialization requirements for the EEPROM input/output microprocessor through to the software alerts for health lane degradation to 'work arounds' for the PCB design that left resistor arrays susceptible to thermal stress. As can be seen, these recommendations and notices did not simply focus on the software requirements issues but covered diverse aspects of the engineering of the EEC systems.



Flight takes off, Launceston airport.
Climb passes 7,000ft ASL.
Thermal cycle induced stress due to differential thermal expansion between printed circuit board and resistor.
20/12/2002: Manufacturer notice on in-flight shutdown & engine restart procedures. Advises of harness installation procedures and review of fault codes and advises interrogation of MCDU every 50 flight hours.
"EEPROM input/output microprocessor is configured with positively charged hexadecimal binary 1s occupying unused memory sections".
17/1/2003: Manufacturer service bulletin SB-BR700-73-101404 software modification for processors communications modules (A3/A4) to change pattern of unused areas of EEPROM from hexadecimal binary 1's to 0's reducing possibility of checksum failure. Compliance on next EEC repair or by manufacturer (not mandatory).
Some areas of unused memory section of input/output microprocessor in Channel A EEPROM become negatively charged with binary zeros.
Assump: Channel B fails from intermittent loss of signal from solder joint fractures to resistors of analogue interface module circuit board
20/12/2002: Manufacturer service bulletin SB-BR700-73-101401 instructions to reduce stress on resistors by attaching them to analogue interface module circuit board and connect them to original solder pad by flying leads. Compliance at next shop visit for EEC repair or by manufacturer (not mandatory).
Assump: checksum failures cause healthlane degradation and Channel A shuts down.
Same resistor pack used on both channels.
Assump: healthlane degradation causes Channel B to shut down.
20/1/2003: Manufacturer service bulletin SB-BR700-73-900316 fleet operators told of inspections and modifications to improve EEC reliability, 10 changes to be made at earliest opportunity without affecting flight schedule. Requires EEC return to manufacturer.
Dual channel failure of right engine electronic engine controller (EEC)
Manufacturer incorporates software upgrade for EEC v7.0, removes "possibility of certain intermittent failures to trigger a 'health lane' degradation without triggering the corresponding maintenance message".
Right Engine EEC removes power from engine fuel control monitoring valve
Engine fuel control monitoring valve spring loaded into closed position.
27/11/2002: Manufacturer issues update on testing of Fuel Metering Unit and EEC
Right engine is starved of fuel.
Right engine uncommanded in-flight shut-down.
25/10/2002: Manufacturer issues warning over in-flight shutdown event and lists warning codes witnessed.

Figure 4: Safety Management Perspective on Electronic Engine Controller (EEC) Failure

Similar comments can be made about the regulatory organizations. Immediately following the first incident, the relevant civil aviation authority directed the aircraft operator to review all fault codes at the end of each day for the ‘occurrence aircraft’ until further notice. This requirement was later modified to include an MCDU review at every airport for which there was engineering support. This incident is unusual because one might have expected the recommended review to have identified precursors to the EEC fault codes that were logged when the second incident occurred (see figure 3). However, the official report avoids any comment about the way in which a second failure occurred even though the civil aviation authority had taken steps to avoid such a recurrence through monitoring the software logs. Following the failure of the left engine, additional requirements were developed to ensure a review of fault codes at the end of each day's flying for all operator aircraft. They also required a review of fault codes after each sector where engineering support was available to include all of the operator’s aircraft of the same type involved in this incident.

In addition to the steps taken by the manufacturer and the regulator, the aircraft operator also reacted to this incident in several different ways. They required that the MCDU was interrogated for EEC faults after each flight into a manned port. Recurring faults would result in the EEC being replaced. All of the associated codes and corrective actions were to be reported to the national civil aviation authority. The EEC PCB’s were modified to reduce the problems created by thermal cycling of the resistors across all of the fleet. Following any modification, the MCDU interrogation was to continue after every flight into an airport with a service engineering capability for two weeks. This period would then be extended to regular service intervals once the reliability of the modifications had been established. All EECs returned to the manufacturer were to be upgraded, using the software modifications mentioned before. The entire fleet was modified by the first quarter of 2004.

The key point here is to illustrate the diverse approaches that were used to address the many different causes of this incident. As noted, any focus on requirements issues must not distract from the wider engineering issues that concern the manufacturer, operator and regulator. These recommendations illustrate another critical point; none of them deal directly with the problems of requirements engineering. It is remarkable how few accident reports ever deal directly with development issues [11].

This can be interpreted in one of two ways. Perhaps, the lack of recommendations dealing with requirements engineering illustrates an important omission on the part of investigatory agencies that are otherwise missing important opportunities to prevent similar failures from affecting future products that are engineered using the same processes. Alternatively, such omissions might reflect the pragmatic view that it is impossible to develop perfect requirements and that it is more profitable to focus on ensuring the immediate safety of existing systems without imposing undue constraints on the processes that might used to guide the development of future systems.

3. Discussion: Biases in Forensic Software Engineering

The previous case illustrates the complexity of engineering failures, which can often be obscured by simple *prima facea* claims that accidents stem from inadequate requirements gathering. Forensic software engineering is also subject to other forms of bias. For instance, many researchers have a vested interest in promoting particular techniques. Hence, it is possible to read articles that are based on counter factual propositions of the form ‘accident A would have been avoided if technique X or Y had been avoided’. Of course, such arguments are non-truth functional. The accident did occur and hence we must make a judgment based on suppositions about what *might* have happened if different requirements engineering practices had been followed.

For the case study, it is difficult to know what evidence could be recruited to demonstrate that the EEC fault code reporting mechanisms would have been improved if the original units had been developed using formal methods or any other approach. Hansen and Gullesen illustrate one approach when they use UML to identify faults that were deliberately injected into a dual channel architecture similar to that described in our case study [15]. However, demonstrating that a failure mode *can* be detected using a requirements engineering technique is quite different from showing that it *would* have identified the problems that lead to particular accidents and incidents. In other words, such demonstrations often suffer from the hindsight bias mentioned in the previous paragraph.

Bias can be interpreted as influences that prevent objective consideration of an issue or situation. These influences can lead to or be reinforced by the use of logical fallacies to support the findings of accident investigations. In particular, they often seem to be used to justify the identification of requirements failure in the aftermath of software related incidents and accidents. For example, the ‘post hoc ergo propter hoc’ fallacy occurs when arguments move from a premise of the form "A preceded B" to a conclusion of the form "A caused B". Requirements engineering takes place in the earliest stages of many development projects. Hence, analysts may incorrectly assume that by tracing the causes of an adverse event into these initial stages, they are also tracing the underlying, systemic causes of an accident or incident. Caspers Jones recognised this when he argued that the root causes of software failure should be traced back to faulty management and quality control practices rather than to requirements processes [16]. It is very difficult, if not impossible, to capture fully all of the competing requirements that software must continue to satisfy during its lifetime, hence we should focus more on the processes that are intended to trap key requirements problems *during* a software project. In this view, our case study incident can be viewed as a success since redundancy prevented loss of life, and the problems with EEC fault code reporting were corrected once the issue had been identified.

Further fallacies can be identified in the reasoning that is used to identify requirements problems in the causes of incidents and accidents. For example 'argumentum ad ignorantium' occurs when a proposition is claimed to be true because it has not been shown to be false, or vice versa. In our case study, we might assume that poor requirements analysis led to the lack of prominent error code reporting for the EEC, because the accident report does not present evidence in support of the techniques that were used by the manufacturer.

We have already met several other fallacies in the opening sections of this paper. For instance, 'dicto simpliciter' relies on sweeping generalizations of the form '90% of all accidents are due to requirements failure'. Similarly, arguments 'ad verecundiam' are based on appeals to authority. For instance, where requirements problems are diagnosed by reference to previous work that is itself not firmly based on empirical observations but on other forms of fallacy, mentioned above.

4 Caveats and Criticisms

A number of important caveats may be raised about the analysis in this paper. In particular, we have conducted a relatively detailed analysis of a single accident. This is a deliberate decision. The intention has been to expose the complex interactions between software requirements engineering and problems in the underlying avionics. Our aim has been deliberately to avoid the high-level statistical surveys that focus on a small number of software-specific problems. However, we would argue that the EEC case study is typical of a much larger class of accidents or incidents. For example, the Ariane 5 incident that was mentioned in the opening sections of this paper stemmed from a very similar interaction between software requirements and the underlying hardware.

There are further differences between our work and the earlier studies of both Beizer and Vintner. These surveys focused more narrowly on the importance of software lifecycle processes on bug reports. In contrast, this paper focuses on the wider causes of accidents and incidents that partly stem from software related problems. These differences hint at a more general proposition. The statistical work, especially the studies by Vinter, demonstrates the importance of requirements failure as a source of bugs that are documented in project reports. However, our analysis of the interplay between requirements failure and other problems in the wider engineering of complex systems suggests that greater attention should be paid to the interaction between these issues as a cause of major failures.

Nuseibeh and Easterbrook have argued that "the demand for better, faster, and more usable software systems will continue, and requirements engineering will therefore continue to evolve in order to deal with different development scenarios. We believe that effective RE will continue to play a key role in determining the success or failure of projects, and in determining the quality of systems that are delivered". Our work confirms this analysis but we would go on to stress two points of difference. Firstly, it seems unlikely that we will ever be able to entirely eliminate the broad class of problems that are being ascribed to failures in requirements engineering. Secondly, if this is the case we must urgently look at the ways in which such failures might exhibit themselves within the engineering of complex systems.

5 Concluding Remarks

A large number of accident investigations have identified the role that inadequate requirements engineering plays in the failure of safety-critical software [11]. These findings have been supported by several large scale surveys of bug reports in safety-critical projects. However, such findings may be misleading. It is difficult to distinguish between failures in requirements engineering and, for instance, inadequate testing or poor design techniques. In consequence, statistical surveys are often undermined by poor inter-analyst reliability.

Further pitfalls arise from the assumption that inadequate requirements engineering is a cause of all software related accidents for which the system fails to meet its requirements. There is a danger of hindsight bias; it is easy to identify problems after an accident has occurred. However, it can be far harder to establish that an accident might have been avoided if alternate requirements engineering processes had been employed.

This paper has used a case study, focussing on the failure of redundant dual channel EEC, to illustrate these arguments. This incident stemmed in part from software engineering problems, in particular the use of positive binary-1 initialisation for the input/output microprocessor EEPROM created potential vulnerabilities. These issues can be traced back to relatively early stages in the development process and can be ascribed to requirements engineering failures, for example in identifying likely environmental factors that would lead to bit-flip errors with such positive encodings. However, a more sustained analysis of the case study helped to show the more complex causes of the incident. Many of these involved hardware issues and have been resolved, including the vulnerability of the PCBs to thermal cycles. Others relate to design factors, such as the lack of diversity in redundant channels.

Our intention has not been to deny the importance of requirements engineering as a cause of software related failures in safety-critical systems. In contrast, the intention has been to provoke further debate within the emerging field of forensic software engineering. In particular, we would urge greater caution in identifying requirements failure as a 'catch-all' cause of adverse events. We would also welcome insights into why so few accident investigation agencies make recommendations about appropriate requirements engineering techniques when so many researchers have focused on these 'causes'.

Acknowledgements

We thank the reviewers and our colleagues for the constructive comments and criticisms made on earlier drafts of this paper.

References

[1] C.W. Johnson and C.M. Holloway, A Technique for Showing Causal Arguments in Accident Reports. *Proceedings of the 23rd International System Safety Conference*, 22-26 August 2005, San Diego, California, International Systems Safety Society, Unionville, VA, USA, 2005.

[2] J. Reason, *Human Error*, Cambridge University Press, UK, 1990.

[3] N. Leveson, Software Safety: Why, What, and How, *ACM Computing Surveys*, Vol. 18, No. 2, June 1986, pp. 25-69.

[4] P.Ladkin, The Ariane 5 Accident: A Programming Problem? Technical Report, RVS-J-98-02, Bielefeld University, Faculty of Technology, 1998.

[5] Peter G. Neumann, editor, Risks-Forum Digest, available at http://groups.google.com/group/comp.risks [visited March 27, 2006].

[6] Kimberly S. Hanks, John C. Knight, Improving Communication of Critical Domain Knowledge in High-Consequence Software Development: an Empirical Study, *Proceedings of the 21st International System Safety Conference (ISSC'03)*, Ottawa, Canada, August, 2003.

[7] UK Health and Safety Executive, Out of Control: Why control systems go wrong and how to prevent failure. Health and Safety Guidance report 238, Bootle, UK, 2003.

[8] O. Vinter, From Problem Reports to Better Products. In L. Mathiassen, J. Pries-Heje and O. Ngwenyama (eds.), *Improving Software Organizations: From Principles to Practice*, Chapter 8, Addison-Wesley, 2002.

[9] D.G. Firesmith, Engineering safety-related requirements for software-intensive systems, *27th Int. Conference on Software Engineering*, St. Louis, MO, 720–721, 2005.

[10] B. Nuseibeh and S. Easterbrook, Requirements Engineering: A Roadmap. In A. C. W. Finkelstein (ed), *The Future of Software Engineering, 22nd International Conference on Software Engineering*, IEEE Computer Society Press, 2000.

[11] C.W. Johnson, A Handbook of Accident and Incident Reporting, Glasgow University Press, U.K., 2002.

[12] B. Beizer, *Software Testing Techniques.* Second edition.Van Nostrand Reinhold, New York, 1990.

[13] Australian Transportation Safety Board, Serious Incident Ref. 200204444, 11 km N Launceston, (VOR), 04-Oct-02, 2003.

[14] J. Austin, R. Davis, M. Fletcher, T. Jackson, M. Jessop, B. Liang and A. Pasley, DAME: Searching Large Data Sets Within a Grid-Enabled Engineering Application. *Proceedings of the IEEE - Special Issue on Grid Computing*, (93)3:496-509, March 2005.

[15] K.T. Hansen and I. Gullesen, Utilizing UML and Patterns for Safety Critical Systems, *5th Int. Conference on UML and its Applications*, Dresden, Germany, 2002.

[16] C. Jones, Patterns of Large Software Systems: Failure and Success, *IEEE Computer*, Vol.28, Issue 3, March 1995.