Software Tragedies: Case Studies in Software Safety

A. J. Kornecki, Embry Riddle Aeronautical University, Daytona Beach, FL
J. Lewis, Lockheed Martin Information Systems, Orlando, FL

Keywords: software failures, accidents, software safety, causal analysis

Abstract

The paper takes a look at the events surrounding true safety failures in software intensive systems. It examines some less well-known software safety events and tries to derive lessons from them. It attempts to identify patterns or similarities in the environment surrounding software accidents in which the loss of human life occurred. It also attempts to reveal what, if anything can be learned from these accidents to prevent similar occurrences in the future.

Three software case studies have been used to set the stage for the paper. They are: the Patriot Missile System failure in the Gulf War in 1991, the London Ambulance Service software system failure in 1992, and the Marine Corps MV-22 Osprey crash in 2000. Each case study covers a separate accident in which software failure has been identified as the primary cause of the accident leading to the loss of human life. Each case study examines the context of the accident as well as the events leading up to the accident. The actual cases have been described elsewhere. These incidents were chosen because, while each happened in a different context, all three happened because of faulty software, and all three have been argued to have led to the loss of human life. These incidents are true software safety failures. The paper attempts to draw lessons from the analysis of these software failures and make suggestions for further improvements in software safety.

Introduction

Is software safety really an issue? Books and papers have been written on the topic making valid arguments that industry needs improvement in this area. The assumption is made that there exist hazardous environments in which software is primarily responsible for ensuring the safety of human personnel. If the software in these environments is written incorrectly, people may sustain injury or death. The goal of literature on the topic is to ensure that hazardous situations are identified and subsequent steps are taken to build software that mitigates the associated risks.

Nancy Leveson in the appendices of her text (ref.1) provides exhaustive analysis of several system failures in the medical, aerospace, chemical and nuclear industries. While software issues are noted, most of the examples focus on general system safety. Robert Glass's text (ref.2) shows what went wrong in several disasters attributed to software. These include the Denver Airport baggage system, the IRS modernization, American Airlines' failed reservation system, New Jersey's Department of Motor Vehicles software failures, the NCR inventory system that nearly destroyed its customers, and the collapsed next-generation FAA Air Traffic Control System.

There are few well-known cases where software was primarily responsible for injury, financial loss, and the actual or implied loss of life: THERAC's radiation problems (ref.3), the self-destruction of the Ariane 5 (ref.4), and controversial accidental shooting down of an Iranian airliner by the U.S.S. Vincennes (ref.5). The purpose of this paper is to examine some less well-known events related to safety of software intensive systems and derive some lessons from them.

Specifically, this paper examines the Patriot Missile System failure in the Gulf War in 1991, the London Ambulance Service software system failure in 1992, and the crash of the U.S. Marines MV-22 Osprey in 2000. While sources addressing these events are scarce, enough information has been obtained to describe the overall story, analyze the causes of failure, and derive some lessons to be learned from the software failures. These incidents were chosen because, while each happened in a different context, all three violate safety due to faulty software, and all three contributed to the loss of human life. These incidents are true software failures.

Patriot Missile System Failure

The Story:

The Patriot Missile System was first developed in the 1960s as a surface-to-air missile (SAM) system that was easily mobile and could operate a few hours in one location before moving to another. Originally built to target Soviet built medium and high altitude aircraft and cruise missiles, the Patriot system was designed to bring down objects traveling around Mach 2. The system has evolved to counteract to short-range ballistic missiles and is well known for its action and successes against Scud missiles in the first Gulf War.

It was in the Gulf War that a serious error in the Patriot Missile System led to the deaths of 28 American soldiers. During Operation Desert Storm a Patriot missile battery was located around an armed forces base in Dhahran, Saudi Arabia. On February 25, 1991 the missile system failed to track and intercept an incoming Scud missile which subsequently hit a barracks, killing 28 Americans.

The Information Management and Technology Division of the United States General Accounting Office carried a formal investigation. The General Accounting Office report (ref.6) traces the source of system failure to a software problem.

Failure Causal Analysis

Each Patriot missile defense system is composed of a several batteries (typically six) that ensure the safety of the protected area. Each battery consists of an independent ground-based Radar Unit, and Engagement Control Station allowing manual or automated missile command and control, eight missile launchers, and a communications station. The majority of the responsibility for missile operations is achieved through the Weapons Control Computer, which tracks and intercepts targets and allows for command and control functions. The design of the computer itself has evolved from its original 1960s configuration but is still based on 1970s hardware and software technology.

In operation, the weapons control computer uses information from the radar to track, identify, and plot the target and then release missiles to intercept the incoming target. Once the computer has identified a target (e.g. Scud missile) using pre-programmed target characteristics (size, speed, shape), an electronic detection device in the radar system called the *range gate* calculates a three dimensional area where it should next look for the target. If the target is then acquired in the predicted range, the computer confirms the target velocity and trajectory.

Figure 1 shows a correctly calculated range gate area:

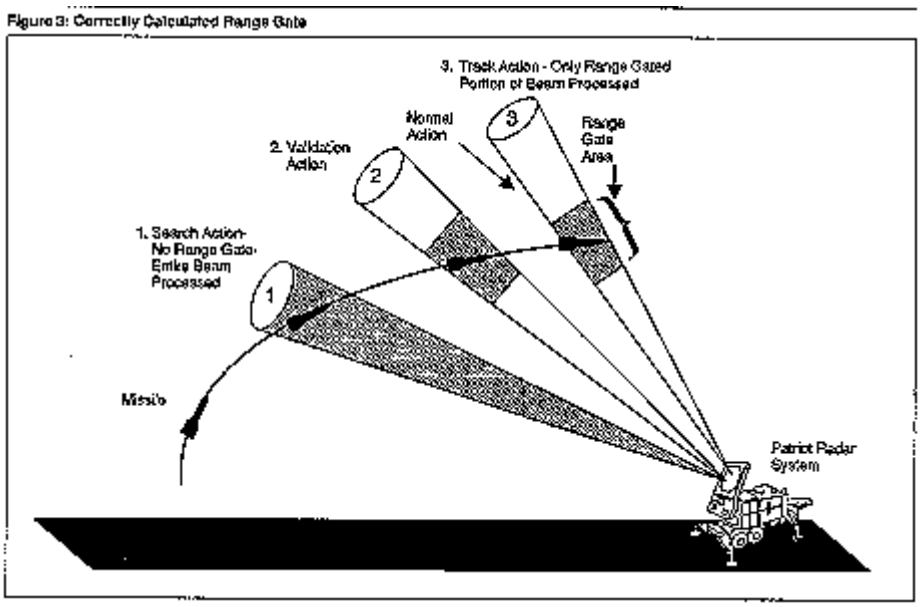

Figure 1 - Patriot Missile Tracking System (ref.6)

The range gate predicts in what area the target will be located based on known pre-programmed characteristics of the missile. Velocity is modeled as a real number. The internal clock keeps time to tenths of a second, and then multiplies by 1/10 to produce the time in seconds. The system computer uses 24 bit fixed-point registers. This means that the non-terminating binary expansion of 1/10 is chopped at 24 bits (0.000110011001100110011001100 in binary). Over time calculations of time based on the truncated value of 1/10 add to become a significant source of error (as shown in Table 1).

Table 1: Patriot Missile System Error Accumulation over Time

Hours	Seconds	Calculated Seconds	Error	Meters Shifted
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0025	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3433	687

Assuming a target traveling at 1,676 meters per second and a Patriot battery operating continuously for 100 hours with an operating timing error of .3433, the range gate will have missed its target area by 687 meters. With this deviation the target will be neither validated nor intercepted.

The Patriot missile battery that failed at Dhahran had been in continuous operation for over 100 hours. Ironically, a software fix had been developed in the United States by Raytheon, the developer, and arrived in Dhahran on February 26, the day after the Scud attack. Also ironically, the situation would have been averted had the system computer been rebooted every 10 hours or so, which is about a minute-long operation.

Lessons Learned

- **Upgrading safety-critical software systems can be dangerous and should involve extensive testing.** Originally developed in the 1960s, the Patriot missile system evolved over time, and in components. The system was originally developed to target aircraft traveling around Mach 2. Through the evolution process, the system was upgraded to target missiles traveling at Mach 5. Unfortunately the system was never tested on these faster moving targets over extensive periods of time.
- **Small faults in safety-critical software can have large, costly consequences.** This is certainly a problem in all software, but must be especially addressed when safety is a factor. At speeds as high as Mach 5 over periods of even a few hours, small numerical errors due to round-offs produce large shifts in predicted missile trajectories.
- **If safety-critical software is migrated, software redesign should be considered.** The original intent of the Patriot system was to provide a mobile defense against aircraft for short periods of time. The accident revealed that the original hardware and software was not effectively migrated to the new environment of a stationary battery tracking Mach 5 missiles. It would have been wise to acknowledge the original intent of the developers and identify assumptions subsequently made in the implementation (i.e. the 24 bit register) so that redesigns could be made where necessary.

London Ambulance Service System Failure

The Story

In the 1980s the London Ambulance Service used a manual system for ambulance mobilization and dispatch services. Central Ambulance Control (CAC) would receive a London emergency 999 call, and the control assistant (CA) receiving the call would write down the call details on a pre-printed form. The incident would be located in a map book with reference coordinates, and the set of information would be sent off on a conveyor belt to a central collection point. A CAC staff member would collect the paperwork and identify which of several resource allocators should be called (North East, North, West, or South divisions of London). Using status and location information for ambulances provided by a radio operator, the resource allocator decides which resource should be mobilized. The resource is then recorded on the form and passed on to a dispatcher who contacts the ambulance directly. The whole process should take less than 3 minutes.

The Finkelstein report (ref.7) identifies some problems with the manual system:

a) identification of the precise location can be time consuming due to often incomplete or inaccurate details from the caller and the consequent need to explore a number of alternatives through the map books;
b) the physical movement of paper forms around the Control Room is inefficient;
c) maintaining up to date vehicle status and location information from allocators' intuition and reports from ambulances as relayed to and through the radio operators is a slow and laborious process;
d) communicating with ambulances via voice is time consuming and, at peak times, can lead to mobilization queues;
e) identifying duplicated calls relies on human judgment and memory and is error prone;
f) dealing with call backs is a labor intensive process as it often involves personnel leaving their posts to talk to the allocators;

g) identification of special incidents needing a Rapid Response Unit or the helicopter (or a major incident team) relies totally on human judgment.

A strong case can be made for a computer aided dispatch system that would address many of these deficiencies. An early attempt was made to computerize the existing LAS Command and Control system but was abandoned after load testing revealed that it could not cope with the demands likely to be placed on it in the London environment. The new plan was to implement a state of the art London Ambulance Service Computer Aided Dispatch (LASCAD) system.

The software requirements specification (SRS) was ambitious. It called for a totally automated system where the majority of incoming calls would result in "an automatic allocation proposal of the most suitable ambulance resource" (ref.7). In complex cases, a human resource allocator would be called upon to identify and allocate the best resources, but for the most part the original CAC staff member receiving the call would see the incident through to completion. In late 1990, work was begun on the SRS transitioning to design in summer of 1991. The LAS management mandated that the entire system, in a single phase-over process, be up and running by January 8, 1992. While concerns were raised about the non-negotiable time frame, a consortium of Apricot, Systems Option, and Datatrak won the contract for £1.1 million. It is interesting to note that the competing contractors were asking around £8 million. Questions were later raised as to why the bid was significantly cheaper than those of the competitors.

After an unplanned phased implementation process over the course of the first nine months of 1992 in which the untested system proved itself to be unstable in each phase, the decision was made to migrate directly to the fully implemented, fully automatic system on October 26, 1992. Pandemonium ensued. The system was lightly loaded on startup, but as the day progressed and the number of calls increased, a build up of emergencies in the system increased. The system bottlenecked in several areas of operation and entered a vicious spiral of cause and effect where the bottlenecks caused more bottlenecks. On October 28, at 11 pm, after two consecutive days of system failure and personnel frustration, the LAS instigated an unplanned manual backup procedure. When the dust settled and the disaster was contained, over twenty would-be patients had lost their lives. While none of the coroners' reports has specifically stated that the LAS can be blamed for the death of a patient, at least one LAS trade union, NUPE, insisted that the untested system has directly led to patients' deaths (ref.7).

Failure Causal Analysis

Like many instances, the system's failures cannot be tied to any single cause. Rather, a multitude of small problems worked together to produce the system's failures. Software alone cannot be blamed for the deaths of twenty individuals in this case, but the software and its lack of quality certainly played a major part.

It is clear that the LASCAD software was not complete or reasonably tested. Data transmission problems existed between mobile data terminals and it was unclear how accurate the Automatic Vehicle Location System (AVLS) software was. Visual Basic, used to write all screen dialogues, was added as a tool some time into the project as it had just been released. Unfortunately the developers, in their haste to finish the project, overlooked the fact that Visual Basic is a good tool for fast development, but not a good idea for use in time-critical, safety-related systems. Visual Basic applications are not fast. The LASCAD users experienced this and started preloading all the screens they needed to use. This in turn put a large demand on the processor and memory not only slowing the process more, but in some instances causing system crashes.

The LASCAD software had not been well tested. The backup file server had not been tested at all. No attempt had been made to foresee the effects of incomplete or inaccurate data to the system (e.g. late status reporting, incomplete vehicle locations). The inaccurate data led to a large number of exception messages being generated. To compound this problem, it seems that when the exception list grew and exceptions were scrolled off the page, there was no way to deal with the now invisible exceptions. This in turn had the effect of producing more exception messages. There was no way to go back to see if the proper vehicle had ever been dispatched to handle the emergency. The exception handling software itself became a major system bottleneck.

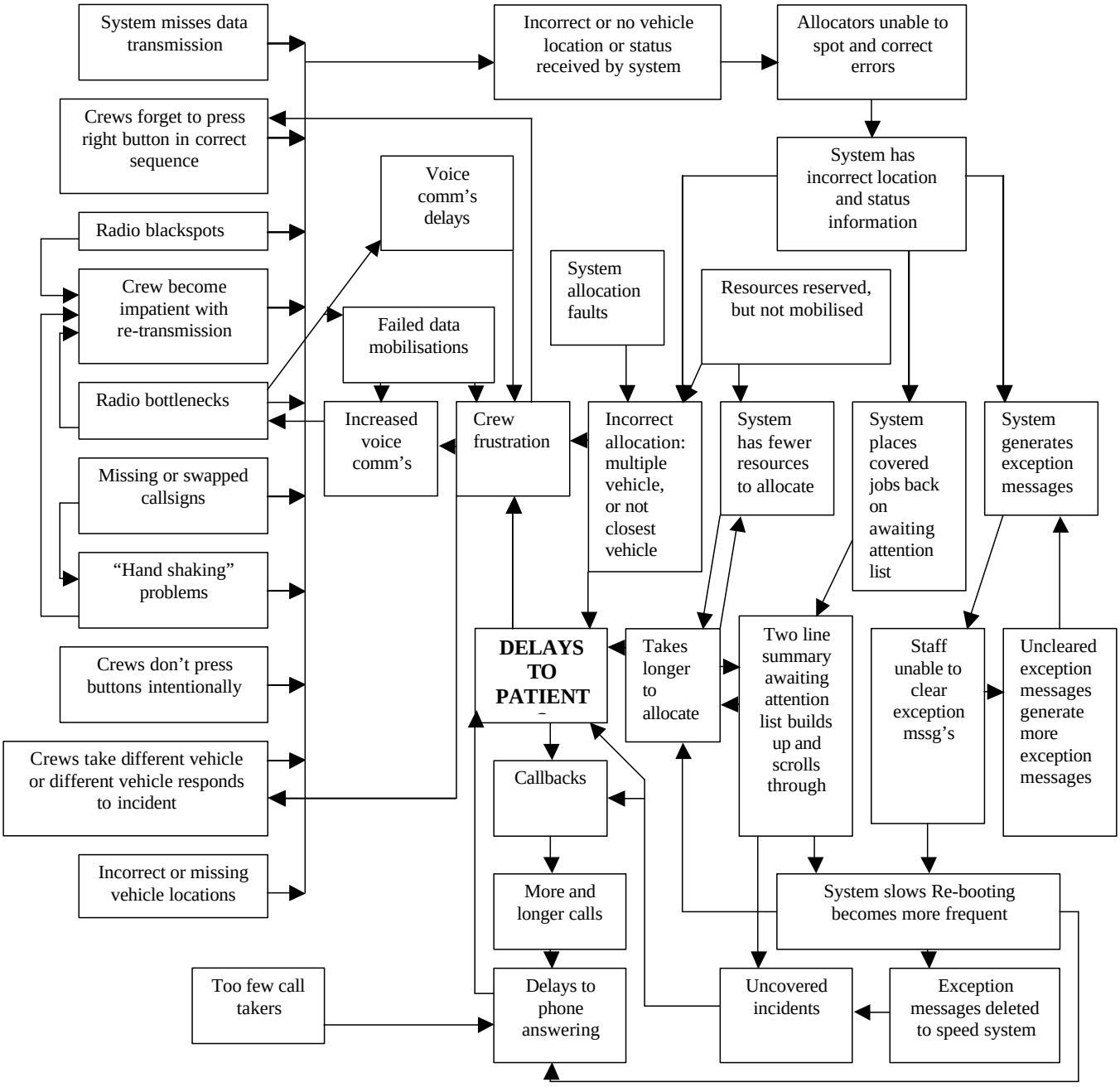

Figure 2: LASCAD Cause / Effect Relationships (ref.7)

It is also clear that the LAS staff operating the system had inadequate training. While the staff and ambulance crews had received some training, it was carried out well before the original planned implementation date. There was a delay of over a year between when the staff was trained and when they actually had to use the system. The training provided was not always comprehensive and often inconsistent, exacerbated by the fact that constant changes were being made to the system and its interface throughout its development.

Lessons Learned

- **Tools and operating systems used in safety-related applications should be well chosen and industry proven.** Using Visual Basic to develop a safety-related application running on Windows 3.0 was not a good idea, especially since safety in this case depended on speed of execution. Running a Windows operating system where speed is a concern is not recommended due to its non-preemptive environment. On top of that, the development team chose an unproven, recently released development tool that was designed primarily for prototyping and developing small, non mission-critical systems.
- **Applications need to be thoroughly tested before introduction to a safety-related environment.** The original plan called for the LASCAD system to be fully implemented on a set date (one would assume after complete testing). Due to the delayed time frame and the target date being missed, it was decided mid-stream that a phased implementation would take place where whatever was finished would be put into operation. During phase-in the system consistently showed itself to be buggy and unstable. Yet the decision was made to continue with the final phase and implementation of the fully automatic LASCAD system.
- **Safety-related software operators need a complete, well-developed training program to equip them for the job.** It is generally recognized that operators of complex safety-critical applications receive intense training. While training was planned and implemented in the LASCAD project, it was too little and much too early. The training was neither consistent nor comprehensive with substantial skills decay between training and the operations. The software and its user interface had been modified between the time of the training and the operational environment. This was evidenced by the LASCAD staff's frustration, overall lack of system confidence, and unfamiliarity with the system's procedures and protocols.

The Marine Corps MV-22 Osprey Crash

The Story

The following is an excerpt from a special briefing given by Marine Corps Maj. Gen. Martin R. Berndt on April 5, 2001 about the MV-22 accident (ref.8):

> *"On December 11th of last year four Marines perished when an MV- 22B Osprey call sign Crossbow 08 assigned to Marine Medium Tiltrotor Training Squadron 204 crashed while on approach to Marine Corps Air Station New River. The aircraft took off from New River at 5:47 p.m. local to conduct a night vision-aided training mission. At 7:17 p.m., after completing its third of four planned radar approaches into the air station at New River, Crossbow 08 made a left-hand turn heading north, accelerated to 180 knots, climbed to 1,600 feet, and converted to the airplane mode. That is, the nacelles, which are these large cowlings on the end of the wing, rotated forward.*

During this portion of the flight Crossbow 08 was in contact with air traffic controllers at Marine Corps Air Station New River. The controllers directed the aircraft to turn to magnetic headings of 280 degrees, 250 degrees, 230 degrees, and finally 200 degrees, or south-southwest. Crossbow 08 acknowledged and executed all of these heading changes. The aircrew used the flight director panel, which can be likened to a programmed autopilot, to complete these turns. During this series of left-hand turns the aircraft's air speed was reduced to 160 knots on the flight director panel, and the nacelles began to transition to the helicopter mode. This transition occurs automatically when the air speed is reduced below 160 knots to compensate for the lift loss from the reduced airflow over the V-22's fixed wing.

At 7:23:40 p.m. shortly after the nacelles began to transition from the airplane mode, a main hydraulic line ruptured that feeds the aircraft's left squash plate actuators. When the flight control computers sensed the problem, they stopped the rotation of the nacelles."

"When the hydraulic line ruptured, the primary flight control system, or PFCS, reset button illuminated, in accordance with published procedures, the aircrew pressed the reset button. This action started a chain of unpredicted and uncontrollable events that caused accelerating and decelerating actions of the aircraft until it entered a stalled condition and departed controlled flight.

At 7:24:10, just 30 second after the failure of hydraulic system number one, Crossbow-08 crashed in a marshy area seven miles north of the airfield in a nose-down attitude."

Failure Causal Analysis

At first glance it appears that the doomed MV-22 flight experienced merely mechanical problems. A hydraulic line responsible for feeding the aircraft's left squash plate actuators (used in providing blade pitch control) ruptured due to chafing by a moving wire bundle. The backup systems worked correctly and the isolation valve stopped the hydraulic fluid leak. The squash plate actuators were still operable though the left actuators were receiving more hydraulic pressure than the right ones. Berndt maintains, *"this hydraulic failure alone would not normally have caused an aircraft mishap"* (ref. 8)

Two other references provide additional information on what brought down Crossbow-08 (refs. 9-10). The hydraulic failure caused a series of warning indicators to go off in the cockpit, one of these being the illumination of the primary flight control system, a light sitting squarely in front of the pilot and copilot, and a warning tone. The operations manual tells the pilot to respond to this warning by pressing the primary flight control system reset button.

The pilot performed as per the published procedure and pressed the reset button. The flight system control software reset the system and attempted to reset both prop rotor pitch angles. Due to the hydraulic failure, asymmetric pressure was applied to the prop rotors. This resulted in un-commanded movements about the pitch, roll, and yaw axes. The cockpit data recorder revealed that the reset button was pressed as many as 10 times during the last 20 seconds of the flight in order to maintain control during the emergency. Unfortunately this action resulted in the aircraft going increasingly out of control as rapid and significant asymmetric changes to prop rotor pitch were made every time the system was reset.

Because the MV-22 is a complex aircraft, the primary flight control system needs to be reset by the system software during many potential emergency scenarios. Resetting the system normally renders the MV-22 more controllable in most emergency situations. In the case of a hydraulic failure to a prop rotor, however, the primary flight control system is not supposed to reset itself due to its potential to apply asymmetrical thrust to the aircraft. The reset button, in this situation, is supposed to do nothing to the primary flight control system.

Bell and Textron, the companies involved in developing the system, had apparently taken the situation into account. There was a published procedure for this type of hydraulic failure in the operations manual, and the pilot followed it. However, the logic implemented in the software did not account for this situation. It is apparent that testing of the software did not reveal this anomaly, and there is a question as to whether this particular situation was ever tested. Naval Air Systems Command (NAVAIR), responsible for inspections and system tests, also missed it.

Lessons Learned

- **All emergencies in which safety-critical software is running should be simulated before introduction to the safety-critical environment.** It is during an emergency that everything that can go wrong will. Often multiple mechanical failures will happen at once, and any operator involved will have his hands full handling the situation. It is critical that during this type of situation, the software is not working against the operator by making things worse. Every emergency situation that the software is responsible for handling should be simulated to prove that the software is working correctly.
- **Complex situations with multiple subsystem interactions involving safety-critical software should be carefully analyzed.** In this case, the interactions between the primary flight control system software and the backup hydraulic system in an emergency situation had not been thought out. It is complex situations like this that are hard to foresee during design, development, and testing. But it is these situations that are the sometimes most safety-critical.

Conclusion

Any accident that ever happens is typically due to a combination of multiple factors. Accidents involving safety-critical software are no different. The Patriot missile accident could have been avoided had there been better communication between the software development team, military commanders, and the missile batteries in the field. The software fix had already been developed but it arrived a day late. A better management team, especially in regards to the development life cycle, could have helped the London Ambulance Service system. The MV-22 accident would not have happened had the military taken care of the hydraulic line chafing problems that had been plaguing the fleet for so long. Yet in each of these cases the faulty software remains a major causal factor in these safety-related failures.

Is the issue of software safety a legitimate concern? While many software failures, large and small, occur daily, and while a considerable number of safety-related failures have occurred recently, only a relatively few software failures have historically been a cause of accidents that resulted in injury or death. As software becomes more prevalent in safety-related systems, these tragedies provide some lessons to learn and a strong reminder that consequences of software failures can be disastrous.

References

1. N.Leveson, Safeware - System Safety and Computers. Reading, Mass: Addison Wesley, 1995
2. R. L. Glass, Software Runaways: Lessons Learned from Massive Software Project Failures. Upper Saddle River, NJ: Prentice Hall, 1997
3. N. Leveson, C.S. Turner, *An Investigation of the Therac-25 Accidents*, IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41 (http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)
4. J. Gleick, *A Bug and a Crash: Sometimes a Bug Is More Than a Nuisance*, http://www.around.com/ariane.html
5. Aviation Safety Network, *Accident Description, Flight IR655*, http://aviation-safety.net/database/1988/880703-0.htm
6. GAO/IMTEC-92-26, *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, United States General Accounting Office (GAO), Information Management and Technology Division, February 4, 1992, http://klabs.org/richcontent/Reports/Failure_Reports/patriot/patriot_gao_145960.pdf.
7. A. Finkelstein, *Report of the Inquiry into the London Ambulance Service* as presented at the International Workshop on Software Specification and Design. The Communications Directorate, Southwest Thames Regional Health Authority, February 1993, http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las/lascase0.9.pdf
8. M. R. Berndt, Marine Corps Maj. Gen. *Briefing on V-22 Accident by Maj. Gen. Berndt*, News Transcript. United States Department of Defense, April 5, 2001, http://www.defenselink.mil/news/Apr2001/t04052001_t405mv22.html
9. C. Bolkcom, *V-22 Osprey Tilt-Rotor Aircraft*, CRS Issue Brief for Congress, November 5, 2001
10. G. Dady, *Osprey to resume flight testing*, NAVAIR V-22 Public Affairs Press Release, March 1, 2002

Biography

Andrew J. Kornecki received MSEE'70 and PhD'74 degrees from the University of Mining and Metallurgy in Krakow, Poland. After teaching a doing research on three continents, currently he is employed as a faculty at Embry Riddle Aeronautical University (ERAU), Daytona Beach, FL. He has been teaching a variety of undergraduate and graduate courses: computer organization, modeling and simulation, real-time systems, performance analysis, software safety. Recently he has been engaged in research on testing and certification issues and assessment of development tools for safety critical real-time systems.

Joshua Lewis received a B.S. in Aeronautical Science from LeTourneau University in 2001 and a Master of Software Engineering from Embry-Riddle Aeronautical University in 2002. He currently works as a software engineer in the field of training and simulation for Lockheed Martin Information Systems in Orlando, FL.