National Aeronautics and Space Administration

SYSTEM FAILURE CASE STUDIES

JANUARY 2012 VOLUME 6 ISSUE 2

Critical Software: Good Design Built Right

INTRODUCTION

The role of software in the control and operation of flight systems has grown dramatically since the landmark Apollo program which ended in 1972. Guidance and navigation systems, science payloads, environmental control systems—all critical—interface with hardware and humans to control almost every significant event in the flight profile. Engineers must ask, "Are we building the right software?" Then, just as important, "Are we building the software right?"

Software assurance and system engineers interestingly note that, while software may display defects in design or implementation (coding), software does not 'fail' after some period of proper operation like how a metal component may fracture. Furthermore, a software defect may not be immediately apparent, but when exposed, the defect will operate exactly as programmed. Examples of defects include: Function (missing requirement), Assignment (incorrect or missing assigned value), Interface (design specifies the interface point to a number, but the implementation points to a character), Checking (missing or incorrect validation of parameters, data, or a test case versus certain measures), Timing (resource serialization is needed and missing, wrong resource serialized, or wrong serialization technique), Relationship (traceability from requirements to test in error or missing), Build (mistake in version control, change management or library systems), Documentation (inaccurate or missing information in documentation), Algorithm (problem affecting task can be corrected by re-implementing an algorithm instead of a design change), Project (team tasked beyond resource capabilities), Verification method (not explained or wrong for situation).

The following ten recent catastrophic mishaps identify the roles that software played as significant causative factors and help recognize the effects of failure and the lessons that can be learned from the mishaps.

MISHAPS WITHIN NASA

STS-126 Shuttle Software Anomaly

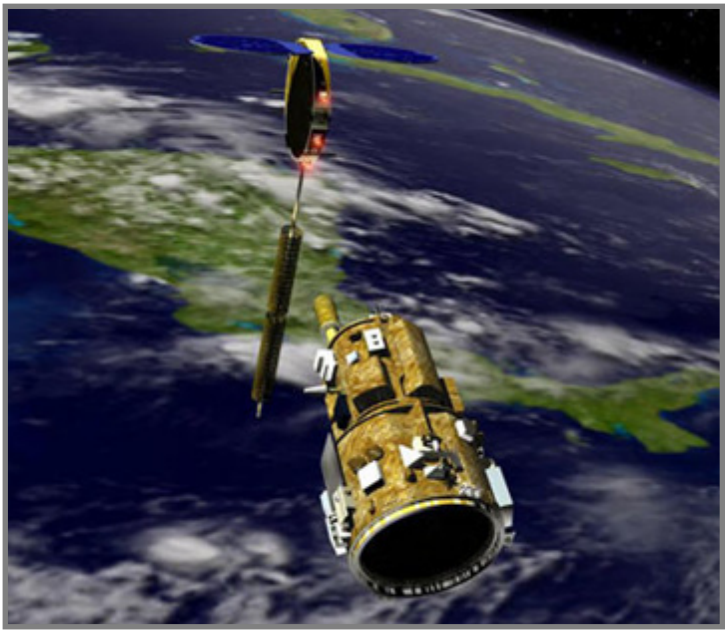
Space Shuttle Endeavour and the STS-126 crew launched on November 14, 2008. Upon reaching orbit, the shuttle-to-ground S band communications used during launch failed to automatically switch to the more powerful Ka band antenna required during orbit. Then, shuttle-payload communication, through the Payload Signal Processor (PSP), failed to automatically switch from its wireless RF link to its hardwired umbilical cable after reaching orbit. Fortunately, mission control was able to manually command switchover of both without obstructing the mission.

Investigation found that a software change had inadvertently shifted data in the shuttle's flight software code. Because of this build defect, the software did not send configuration commands to the shuttle's Ground Command Interface Logic, and several automated functions failed.

DART Failed Autonomous Rendezvous Test

The DART (Demonstration for Autonomous Rendezvous Technology) launched on April 15, 2005 and was programmed to rendezvous with the MUBLCOM (MUtiple-path Beyond Line-of-sight COMmunications) satellite via close range maneuver without ground control commands. Approximately 11 hours into the mission, DART collided with MUBLCOM. DART transitioned to its departure and retirement sequence without accomplishing any of the 14 mission-critical technology objectives and its fuel supply was depleted.

During the close-range maneuvers, the navigational control for DART failed to transition completely to the Advanced Video Guidance Sensor (AVGS) to calculate its velocity and position relative to MUBLCOM. This interface defect allowed DART to approach MUBLCOM without accurate ranging information. The collision avoidance system operated as designed, but using the same



inaccurate position and velocity information, DART collided with MUBLCOM.

The premature retirement of DART occurred due to recurring computational resets of its estimated position. DART's thrusters fired more often than planned following the resets, attempting to correct for each new estimated position. An incorrect velocity measurement was introduced into software during each reset. This algorithm defect had been discovered prior to the mission but no software change was implemented.

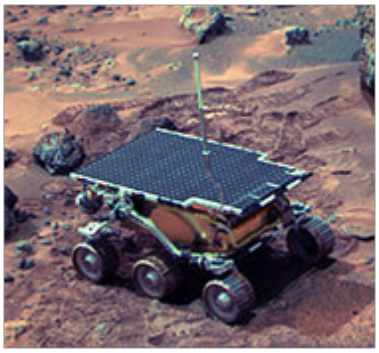
Loss of Communication with the SOHO Spacecraft

The SOHO (Solar Heliospheric Observatory) spacecraft was launched on December 2, 1995 to study the Sun from its deep core to the outer corona and the solar wind. SOHO succeeded in many of its goals, earning it multiple mission extensions. The first came in 1997 when software modifications were uploaded to conserve operation of its gyroscopes. Then in 1998, another software modification caused SOHO to lose lock on the Sun and immediately triggered alarms that sent the spacecraft into an emergency altitude control mode. During recalibration, the spacecraft spun out of control and its altitude had diverged so far beyond control that all power, communications, and telemetry signal were lost.

Ground operators found that a build error in the code modifications had triggered the initial alarm. Critical errors in the code modified to conserve gyro usage configured the gyros incorrectly, and caused inaccurate thruster firings which progressively destabilized the spacecraft. It took the ground team over 5 months to find and completely recover SOHO.

Mars Pathfinder's Unexpected System Resets

The Mars Pathfinder landed on Mars in July 4, 1997. It returned an unprecedented amount of data and outlived its primary design life. But a few days into the mission, the lander's onboard computer repeatedly reset itself, slowing the flow of research data to Earth.



A timing defect in software application code caused the computers to continually reboot. A small low-priority task was unable to complete its function during data flow to Earth when all the high-priority tasks were moving at high rates. A fail-safe mechanism in the software, which resets the system automatically when any performance is interrupted, rebooted the system when the glitch occurred. NASA solved the problem by raising the priority of the task involved and adjusting the priority of other tasking in the code.

Non-NASA Mishaps

Air Traffic Control Communication Loss

On September 14, 2004, the Los Angeles Air Route Traffic Control Center lost all radio communication with 400 airplanes they were tracking over the southwestern United States. The incident disrupted about 800 flights across the country. The Center's main voice communications system shut down unexpectedly.

A function defect was found in a software upgrade to a subsystem for the Voice Switching and Control System (VSCS). The upgrade used a timer to send built-in test queries to the VSCS. The upgrade's test counter counted tests down from the highest number that the system's server and software could handle, counting to zero or until reset. Procedures required a technician to reset the voice switching system every 30 days. The upgrade had a design defect—it was programmed to shut down the VSCS without warning if the reset had not been done, which it did when the counter reached zero after 49 days.

The FAA later implemented a software patch that periodically reset the counter without human intervention.

Widespread Power Outage in the Northeast

On August 14, 2003, high-voltage power lines in northern Ohio brushed against some overgrown trees and switched off. This triggered a distribution system load imbalance and cascading power outage throughout parts of the Northeastern and Midwestern United States and Ontario, Canada. The blackout affected an estimated 55 million people for 2 days, which contributed to at least 11 deaths and an estimated cost of $6 billion.

Computer systems available to local power system operators were designed to issue alarms when detecting faults in the transmission or generation system. Due to programming errors, these alarms failed. System operators were not able to take steps that could have isolated utility failures because the data-monitoring and alarm computers were not working. Without the knowledge of line failure, operators could not act to balance system loads and contain the failure.

A "race condition" or software timing defect was found to be the primary cause of the grid event alarm failure. After the alarm system failed silently, the unprocessed events started to queue up and crashed the primary server within 30 minutes. This triggered an automatic transfer of all applications, including the stalled alarm system, from the primary to the backup server, which likewise became overloaded and failed. Hardware, software and procedural improvements followed to prevent recurrence.

MV-22 Osprey Crash

On a routine training mission on December 11, 2000, an MV-22 Osprey carrying 4 Marines crashed in a wooded area north of Jacksonville, North Carolina, killing all on board.

It was found that the mishap was caused by a burst hydraulic line in one of the Osprey's two engine casings. This coupled with a software defect causing the aircraft to accelerate and decelerate violently and unpredictably when the pilots tried to compensate for the hydraulic failure. The Marine Corps report called for a redesign of both the hydraulics and software systems involved.

Korean Air Flight 801 Crash

On August 6, 1997, Korean Air Flight 801 approached for landing at the Guam International Airport. Heavy rain and reduced visibility dictated the crew to fly a precision instrument approach using the ILS (Instrument Landing System). Air traffic controllers advised them that the ILS glideslope equipment for the available runway was out of service. This required the crew to monitor altitude via altimeter only until the runway was in sight or until minimum approach altitude reached. The aircraft struck a hill approximately 3 miles short of the runway, at an altitude of 660 feet. Of the 254 people on board, 228 were killed; the remaining 23 passengers and 3 flight attendants survived the mishap with serious injuries.

The investigation found that, while the crew failed to properly follow the approach procedure, a ground warning system could have alerted controllers to the unsafe descent. However, a Federal Aviation Administration (FFA) software change limited spurious alerts. The build defect rendered the system "almost completely useless," preventing approach controllers from warning Flight 801 of its premature descent into the hill.

Ariane 5 Failure Forty Seconds After Lift-Off

The Ariane 5 is an expandable launch system used to deliver payloads into geostationary transfer orbit or low Earth orbit. On June 4, 1996, its maiden flight ended in failure, with the rocket veering off its flight path and self-destructing at about 40 seconds after initiation of the flight sequence, at an altitude of about 12,000 feet.

It was found that the failure was caused by complete loss of trajectory guidance due to malfunction in the control software. Software to align the Inertial Reference System (IRS) had been reused from the Ariane-4 system, but behaved differently in the Ariane-5. The software was not properly tested for the trajectory characteristics of the new vehicle. Failure in converting 64-bit floating-point number to a 16-bit signed integer caused an overflow condition (the 64-bit input value was outside the range that could be handled by the 16-bit signed integer). The assignment defect shut down the primary inertial reference system; when control was passed to the identical secondary inertial, it predictably suffered the same fate. Ironically, the code containing the error had been designed for an Ariane-4 launch requirement not shared by Ariane-5 and could have been eliminated. The onboard computer misinterpreted diagnostic data as proper flight data (another software defect) and commanded an abrupt maneuver that ripped the boosters from the launch vehicle and activated the rocket's self-destruct mechanism.

Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile, which killed 28 soldiers.

The all-weather tactical air threat defense at Dhahran contained a software flaw in the system's weapons control computer. This function defect led to inaccurate tracking calculations that worsened over time of operation. At the time of the fatal Scud attack, over 100 continuous hours of operation increased the error to the degree that the Patriot system could not track or intercept the threat.

Patriot had never before been used to defend against Scud missiles, nor deployed to operate continuously for long periods of time. Two weeks before the incident, Army officials received Israeli data reporting loss in accuracy after 8 consecutive hours of operation. While the Army modified the software to correct the defect, the software patch did not reach Dhahran until the day after the Scud attack.

For Future NASA Missions

In these examples, the software did not perform as intended. One driving objective of verification and validation is to ensure the system software behaves as expected under adverse conditions. NASA's approach to validating and verifying system software entails acquiring evidence to answer three questions:

1) Does the system software behave as expected?

2) Does the system software not do what it is not supposed to do?

3) Does the system software behave as expected in the presence of adverse conditions?

A system driven validation and verification approach effectively stresses the system, testing an integrated system under various anomalous conditions as early in development as possible to understand and generate evidence per the three questions above.

At a minimum, a common lesson to be learned is to ensure that project system and software development efforts consider the Verification and Validation three perspectives (NASA's IV&V 3 questions) and employ a system/software validation and verification effort that does the same.

If nothing else, dedicate resources toward validating software requirements—ensure the design is right; for changes made after technical baselines are achieved, a revalidation from a systems perspective is warranted. Where lives are at stake, even given exhaustive testing, design-for-failure considerations would have helped the MV-22 Osprey software deal with hydraulic system damage. Engineering designs should handle faults intelligently—so that persistent faults of the same type cannot bring down a system. Our designs and ultimately our systems need to be more adaptable to their environments.

The backup system for the Air Route Traffic Control Center failed because it lacked a feature to reset its test counter. Question 3 of verification and validation efforts, to ensure system software behaves as expected in the presence of adverse conditions is a mitigation strategy for many of these failures.

Design also failed the power system operators who could not use their backup systems during the Northeast blackout in 2003. Redundant hardware can serve against mechanical failure modes, but supposedly redundant systems reliant on identical software programs (like both Ariane 5 inertial reference systems) can face common-cause failures. A validation of requirements may have shown the importance of "features" like these and illuminated systemic risks to decision makers if requirements were not corrected.

Workarounds stemming from software design shortfalls should be unacceptable to designers, especially for mission critical functions or procedures. The ability to identify when a workaround impacts critical system aspects demands a systems perspective. That someone needed to manually reset the clock of the FAA air traffic control VSCS system was a precursor of failure that hindsight reveals to us. It's true that workarounds for the SOHO and Mars Pathfinder missions salvaged mission objectives. But simulations using an integrated system (even if emulated components are used) that exhaustively run combinations of task thread priority can uncover such problems before critical systems are operated.

When a software design is reused in a different system, it is important to revalidate design and test operation. Investigators did not tell us why all Ariane 4 navigation software features were retained for Ariane 5, except to 'be consistent.' This intent does not recognize that Ariane 5 hardware and requirements were different. Again a systems understanding and a systems verification and validation effort can mitigate the concerns of reusing (or salvaging) previously built designs.

To facilitate good engineering practices and increase the likelihood for success, development efforts must ensure proper controls are in place when changing, modifying, or upgrading safety-critical systems. Any changes, big or small, made to the software must be properly evaluated, assessed and documented, especially changes to safety-critical software that may affect the overall performance of the entire system and threaten life safety. Change must be treated with the same degree of attention as original development. This lesson is evident in the software error found in STS-126 that disabled two automatic functions. An update to the shuttle's flight software code was not properly implemented nor verified leading to the anomaly. Another example is the mishap of Flight 801 where the Minimum Safe Altitude Warning (MSAW) system was inhibited without fully realizing the negative consequences. A third example is the loss of contact with the SOHO spacecraft where critical errors in code modifications to conserve gyro usage destabilized the spacecraft. Good decisions about change are informed decisions; if project managers are truly shown the negative impacts on the system, then they can make well-informed risk decisions. Decision makers need the "system impact" perspective but do not always receive it.

How much and what kind of testing is enough? It's easy to say that extensive (expensive) testing should be performed. It is realistic to put forth a robust testing strategy that adequately stresses the system under adverse conditions, at every level, from unit through system test, using authentic operational and exception scenarios. This is what validation and verification and assurance-related activities focus on. All the tasking in the software code of the Mars Pathfinder was exhaustively tested before the mission, but not all code was tested at once. During the operation when the spacecraft was bombarded by low and high priority tasks, it significantly affected the system's performance. The DART project began as a relatively low-cost, high-risk effort with less rigorous software testing requirements than projects with higher expectations of success. When DART's mission objectives gained importance, requirements to better ensure good software design did not follow. Technology demonstrator projects or projects with low-TRL (Technology Readiness Level) gain much from verification and validation and assurance-related activities. Stressing the system and testing off-nominal conditions is beneficial for a project with less schedule and budget resources to cover these aspects.

For safety-critical systems, it is important to test, not only for what the software should do, but also for what it should not do. Ensure the system can return to a safe state after experiencing a specific negative occurrence, instead of simply allowing itself to reboot or shut down.

Software should be tested over several days of equivalent mission time to find problems such as timing errors or overrunning counters. This would have found the glitch that would have prevented the Patriot to decrease its efficiency when it was operated continuously for over 100 hours.

Summary

The incidents covered in this report not only led to the loss of time and money, but also the loss of life; proof that software plays an immense role as a causative factor to project failures in NASA and industry.

Systems engineering and project management need to integrate validation and verification efforts into systems integration strategies to inform decision making by evidence of what the system software actually does, what the system software does not do, and how the system software behaves under adverse conditions.

To ensure good design and implementation, a proven, successful option available to engineers is to run a software Verification and Validation project parallel to software development. A Verification and Validation project employing reviews, static and dynamic analysis, testing, and formal methods can ensure software conforms to requirements and expected operational behavior at every phase of the project's life cycle.

References

1. Dr. Nancy Leveson, "A Systems-Theoretic Approach to Safety in Software-Intensive Systems," January 2004

2. Marcus S. Fisher, "Software Verification and Validation, An Engineering and Scientific Approach," November 2006

3. NASA Safety Center Special Study, "Shuttle Software Anomaly," April 2009

4. NASA System Failure Case Studies, "Fender Bender," September 2008

5.NASA's DART Mishap Investigation Results, 2006

6. NASA System Failure Case Studies, "The Million Mile Rescue," November 2008

7. Final Report of SOHO Mission Interruption Joint NASA/ESA Investigation Board, 1998

8. Military and Aerospace Electronics, "NASA Tackles Pathfinder Software Glitch," September 1997

9. The New York Times, "Air Control Failure Disrupts Traffic," September 15, 2004

10. IEEE Spectrum, "Lost Radio Contact Leaves Pilots on Their Own," November 2004

11. NASA System Failure Case Studies, "Powerless," December 2007

12. Great Northeast Power Blackout of 2003

13. CNN, Marine MV-22 Osprey Crashes During Routine Training Mission," December 12, 2000

14. Congressional Research Service, "V-22 Osprey Tilt-Rotor Aircraft: Background and Issues for Congress," March 10, 2011

15. NTSB, "Abstract on Korean Air Flight 801 Conclusions, Probable Cause, and Safety Recommendations," 1997

16. Inquiry Board, "Ariane 5 Flight 501 Failure Report," July 1996

17. ESA Press Release, July 23, 1996

18. GAO Report, "Patriot Missile Defense – Software Problem Led to System Failure," February 4, 1992

19. The New York Times, "U.S. Details Flaw in Patriot Missile," June 6, 1991

Questions for Discussion

- How does your organization acquire the evidence to understand that your system software will do what it is supposed to do, under adverse conditions, and won't do what it is not supposed to do (guard against emergent behaviors)?
- How does your organization track configuration management and evaluate change from a systems perspective?
- If your primary unit failed due to software errors, will it cause the same failure in your backup? What is your proper level of redundancy?
- Has the risk level of your project decreased, and your software testing plan increased to drive down risk?
- Do you have contingency plans for on-orbit anomalies? What anomalies have been tested for?
- How does your organization verify reused or modified code?

SYSTEM FAILURE CASE STUDIES

NSC

NASA SAFETY CENTER

Responsible NASA Official: Steve Lilley

steve.k.lilley@nasa.gov

Thanks to Marcus Fisher and Kenneth Vorndran an insightful peer review.