Historical Aerospace Software Errors Categorized to Influence Fault Tolerance

Lorraine E. Prokop
NASA, Langley Research Center
1 Nasa Drive, Hampton, VA 23666
lorraine.e.prokop@nasa.gov
lore.prokop@gmail.com

Abstract*—Since the first use of computers in space and aircraft, software errors have occurred. These errors can manifest as loss-of-life or less catastrophically. As the demand for automation increases, software in mission or safety-critical systems should be designed to be tolerant to the most likely software faults. This paper categorizes a set of 55 historic aerospace software error incidents from 1962 to 2023 to determine trends of how and where automation is most likely to fail, behaving unexpectedly. A distinction between software producing unexpected (erroneous) output versus no output (fail-silent) is introduced. Of the historical incidents analyzed, 85% were from software producing wrong output rather than simply stopping. Rebooting was found to be ineffective to clear erroneous behavior, and not reliable to recover from silent failures. Error origin was within the code/logic itself in 58% of cases, 16% from configurable data, 15% from unexpected sensor input, and 11% from command/operator input. A substantial forty percent (40%) of unexpected software behavior was indicated by the *absence* of code, arising from unanticipated situations and missing requirements, and 16% of incidents were subjectively deemed "unknown-unknowns". No incidents were found to be the result of programming language, compiler, tool, or operating system; and only sixteen percent (16%) of all incidents were considered errors traditional computer science/programming in nature. These findings indicate that for fault tolerance, erroneous automation behavior must be a primary consideration especially at critical moments, and reboot recoverability may not be viable. Special care should be taken to validate configurable data and commands prior to use. "Test-like-you-fly", including hardware-in-the-loop combined with robust off-nominal testing should be used to uncover missing logic arising from unanticipated situations not covered by requirements alone. This study uniquely focuses on manifestations of unexpected flight software behavior, independent of ultimate root cause. We characterize software error behavior and origin to improve software design, test, and operations for resilience to the most common manifestations, and provide a rich dataset for further study.

TABLE OF CONTENTS

1. INTRODUCTION

THIS paper explores incidents of unexpected automation/software behavior and software errors primarily in aerospace to identify and raise awareness of erroneous software manifestation trends. We study a dataset[1,2] of 55 incidents to expose trends in software behavior. We explore if software more often behaves unexpectedly, producing erroneous output, versus simply stopping/crashing. We identify where within the software architecture the error originated – within code itself, within configurable data, or from sensor or command input. We quantify cases of missing code, which includes missing requirements, and quantify "unknown-unknowns" as causes for unexpected software behavior. We also quantify errors resulting from traditional computer science issues or poor programming. By understanding how and where software is most likely to fail, systems may be proportionately better designed, tested and operated for robustness against the most probable failures, and backup strategies may be better architected to detect and mitigate software risk.

Paper Outline

The introduction covers previous work and related background topics such as software standards, common-cause failure, and backup strategies. Section 2 introduces the incident dataset. Section 3 describes the categories chosen for analysis. Sections 4, 5, and 6 provide data, results and conclusions, respectively.

Previous Work

There has been some previous work enumerating failures in space systems, most notably Harland and Lorenz's book, "Space System Failures", published in 2005 [3], which provides detailed accounts of incidents across engineering disciplines. However, more incidents have occurred since then, and no publication thus far has focused solely on software, regardless of root cause, to characterize in more detail how and where aerospace software is most likely to misbehave.

Some notable previous work regarding N-version programming by Brilliant, et. al[4] and software error source by Lutz[5] should be considered regarding backup strategies and in conjunction with the results presented herein. After producing multiple programs *from the same specification*, Brilliant determined that the "cases in which more than one [version] failed was substantially more than would be expected if they were statistically independent.", and that "use of different programming languages … would not have a major impact in reducing the incidence of faults". Lutz found that safety-related software errors are shown to arise most commonly from requirements. These results coupled with this study showing that 40% of errors were due to missing code implies that if dissimilarity is employed, it should not only start with different requirements to minimize this dependency, but also carry the independence through verification to better expose the details of what may be missing. Similarly, no incidents in this dataset were due to compiler, programming language, operating system, or development tool errors, which agrees with the Brilliant study that independence of these may be of lesser value than independence of requirements and test. A more detailed examination of previous work is found in Prokop [2].

A Note on Software Standards

Several aerospace standards exist to ensure quality software. At NASA, NPR 7150.2 "Software Engineering Requirements" [6] standard is used to govern the development of all NASA software, and the NASA Human-Rating Requirements for Space Systems [7] governs design for fault tolerance. Similarly for Aircraft, DO-178C "Software Considerations in Airborne Systems and Equipment Certification" [8], is the guide governing the software lifecycle. Developing software according to these and other standards supports a disciplined and rigorous software development process, builds confidence in software, and provides a framework for understanding software risk. However, even with best practices, we show that software errors occur for reasons beyond process.

Software Common-Cause Errors and Backup Strategies

Although not specifically studied here, the notion of software errors being "common-cause" should be considered, because many, if not all these incidents could be considered common-cause. Software "common-cause" or "common-mode" failures arise when software fails, either erroneously or silently, but because identical software may be duplicated on multiple redundant computers running at the same time, a single software error can affect all redundant computers in the same way simultaneously. This is a software common-cause error. System software architecture determines the vulnerability to software common-mode failures. In systems where there is only one copy of flight software, a single software error could be considered a common-cause error. Mitigating software common-cause error effects should be assessed based on system criticality and time-to-effect. Some common mitigation strategies include providing manual backup to automation (crew or ground control), employing a dissimilar software backup, installing a separate safety monitor for detecting erroneous behavior, failing into a safe mode for communications and power generation, patching the software during flight, and rebooting. Like including avionic redundancy to protect against unexpected byzantine-type faults [9], software backup strategies should be designed to protect for errors in higher-level software control. Determining which of these strategies, if any, were employed or could have mitigated each of these incidents is left to further study.

2. Incident Dataset

A dataset enumerating 55 historical incidents is analyzed and characterized. It includes all incidents found since the beginning of employing computers in aerospace to present day such that the software/automation behaved unexpectedly and possibly could or should have been written differently in hindsight to affect a different outcome. Due to length, detailed accounts of each incident are omitted from this paper, but details of each incident along with comprehensive references for each may be found in [1,2], or obtained in spreadsheet form from the conference/author. In each incident, the automation controlling the system either acted unexpectedly or failed to act (for whatever reason) leading to loss of life, loss of mission, loss of time/revenue, or presented a significant close call. It is important to note that the ultimate root cause of these incidents is not necessarily "software". In fact, it could be argued that in all of these cases the software performed exactly as programmed. Determining root-cause of these failures – identifying *why* the software was programmed that way – is left for further study but may include examples such as lack of system understanding, unknown physics, lack of time or resources, lack of skills, or procedural/process errors. Regardless of root cause, however, the software in these cases behaved in an undesired way to cause an unwanted outcome. Two NASA assessment teams over two years, cited in acknowledgements, worked to identify, assess, and characterize these incidents studying publications, NASA mission archives, aviation reports, books, journals, and verifiable internet sources, so the dataset therefore represents all that the teams could collectively find with credible reference documentation.

Table 1 shows a breakdown of studied incidents by industry and Table 2 shows the impact of these incidents. Ninety percent (90%) of these incidents are in aerospace (spacecraft, aircraft, launch vehicle, and missile combined), with others included as well-known representative software incidents in medical or commercial. As shown in Table 1, over half of the dataset consists of spacecraft. Spacecraft and launch vehicles combined comprise two-thirds of the incidents. Table 2 shows the resultant impact of the software errors, 15% being loss of life or injury, 35% loss of vehicle/mission, 15% premature end of mission, 22% close calls for loss of life or mission, and 7% each for delayed objective or loss of service.

Table 1. Industry of Incidents Studied

Industry	Percent	Quantity
Spacecraft	56 %	31
Launch Vehicle	15 %	8
Aircraft	15 %	8
Missile	4 %	2
Medical	5 %	3
Commercial	5 %	3

Table 2. Impact of Incidents Studied

Results/Impact Summary	Percent	Quantity
Loss of Life	13%	7
Persons Injured	2%	1
Loss of Vehicle/Mission	35%	19
Premature End of Mission	15%	8
Close Call for LOC/LOM	22%	12
Delayed Objective	7%	4
Loss of Service	7%	4

3. CATEGORIZATION DISCUSSION

Erroneous vs. Fail-Silent

First, we make a distinction between software failing "erroneously", which includes the automation producing wrong or unexpected output, and software failing "silent", providing no output at all (i.e., crashing), or significant lag. This is an important distinction because detecting the "fail-silent" case is usually more straightforward. A watchdog timer can detect the fail-silent case. Rebooting is typically used to recover from a silent computer, but the effectiveness of this strategy is unreliable as discussed in Section 5 Results.

Detecting and responding to the "erroneous output" case, however, may not be as straightforward. If a human is onboard, or a ground team is actively monitoring, they may be able to recognize software performing unexpectedly and override the automation to take appropriate action. But if there is no human in the loop, or if time-critical, software/automatic backup systems may be employed to detect/recognize and respond to the primary software behaving unexpectedly. Fail-down strategies should be employed in safety-critical systems to mitigate the effects of erroneous output by transitioning to backup strategies.

Reboot Recoverability

A common strategy to recover from faulty software is to reboot. Unfortunately, reboots do not fix all software problems. The incident dataset was reviewed subjectively considering the following question, "Would reboot have cleared this problem?" A yes/no answer is tabulated and presented under Section 4 Data. This is important to know because depending on the problem, it is often assumed that performing a simple reboot may correct the problem. But given the effectiveness presented here, depending on the criticality, and alternate approach should be considered.

Absence of Code

An interesting statistic studied against this dataset is whether the incident could have been avoided by adding code (in hindsight). The incidents were reviewed subjectively considering the following question, "Could the problem have been averted by adding some code?" A yes/no answer to this question is tabulated in Section 4. It is well understood that it is much easier to know what code to add after a mishap rather than predicting the failure in advance. Considering whether the code could or should have been there is a more difficult question addressed in the categorization of "unknown-unknowns" below. But simply determining if an incident was the result of the absence of software has large testing implications. If software is only tested against requirements, or tested against code that exists, then how can errors caused by the *absence* of software be discovered? Performing off-nominal testing and using random input sets may help uncover missing code. Test campaigns should consider testing both existing code to expose the absence of code proportionate with how errors usually manifest.

The question of absence of code is also closely related to missing requirements and includes "unknown-unknowns" discussed later. It could be said that missing code equates to missing requirements, so in that regard, 40% of these errors would be attributable to the requirements phase. However, the author's opinion is that software requirements are in practice far less detailed than could have been written in advance to avert many of these errors. For example, consider the 2007 F22 first deployment international date line incident. Should there have been a requirement stating, "Software calculations shall consider crossing the international date line.", or alike, or would that error have been better discovered in a test-like-you-fly scenario?

Error Location

A categorization as to where in the software the error originated or initially manifested is performed by distinguishing between the following four groups: code/logic, data, sensor input, and command input. The reason for this distinction is because assuring integrity in each of these areas both pre-flight and operationally have different testing characteristics and procedural validation methods.

First, "coding/logic" includes errors that are in the code itself, encoded into logic or algorithms. This category largely encompasses both the "absence of code" and "computer science/poor programming" categories as discussed and includes missing requirements or logic unable to handle unforeseen circumstances. Next, "data" includes those errors due to misconfigured data, or erroneous stored parameters. This is separated from "code/logic" to distinguish between the fact that software is becoming more data-driven, and that data is more likely to change than the code itself. Also, in data driven systems, even though the code may not change, data may change from flight to flight and is often governed by different validation practices. The third category, "sensor input", addresses errors stemming from unexpected or erroneous sensor input. This distinction is made because testing with actual sensor hardware or generating off-nominal tests specifically targeting random sensor input may help to avert this error. The final category, "command input", includes erroneous command input due to operator or procedural error. These errors should normally be averted through command verification during operations prior to their issuance and by process assurance. The overall prevalence of each of these categories is given in Section 4.

Computer Science / Poor Programming

Since this paper focuses on "software errors" it might be presumed that all these incidents were caused by errors within the realm of the computer science discipline or as a result of poor programming. This category quantifies the incidents subjectively considered to fall within the realm of traditional computer science. This includes errors relating to real-time or concurrent programming, race conditions, priority inversion, or simple programming mistakes such as erroneous keystrokes/keywords. This category also would also include errors introduced through programming language selection, development tools, compilers, or operating systems; however, as our results indicate, no incidents were attributable to these.

Unknown-unknowns

The last category, "unknown-unknowns", a term popularized by Donald Rumsfeld referring to "the ones we don't know we don't know [10]", is a highly subjective category and is largely a subset of the "Absence of code". This attempts to conservatively quantify how many of these incidents arose from knowledge only realized or conceived in hindsight that could not have been discovered ahead of time with reasonable effort. It primarily includes cases where aerodynamics or physics were studied but not fully understood, cases of highly unusual sensor input, or behaviors resulting from an unanticipated situation or created by fault situations. It could be argued that with infinite resources, all of these could have been known, such as by performing more wind tunnel testing, more simulation, more analysis, deeper fault level scenario study, or longer and more robust sensor characterization. A subjective evaluation of the question "Could/should it have been reasonably known?" within reasonable project constraints is provided here. This may be used as a rough level-of-risk measure for the unplanned and unexpected in addition to the more concrete "absence of code" category, and its mitigation should be assessed in relation to software criticality and backup options.

4. DATA

Table 3 shows the tabular data resulting from team analysis of each incident in the dataset according to the categories outlined in Section 3. Erroneous versus Silent is designated with an "E" for Erroneous or "S" meaning "Fail Silent". Yes/No answers are provided for the questions of "Would it have been recoverable by reboot", "Could adding code have corrected this issue", "Could this be considered an unknown-unknown", and "Was this poor programming/computer science discipline issue?" according to the previous discussion. For Error location, "C" is used for "Code/Logic", "D" is used for data, "O" is used for Command/Operator Input, and "S" is used for "Sensor Input".

Table 3. Incident Categorization

Year	System	Title	Result	Erroneous or Silent	Reboot Recoverable?	Missing Code?	Error Location	Computer Science?	Unknown-unknown?
1962	Mariner 1 Mission – Atlas-Agena	Programmer error in ground guidance veered launch vehicle off course	Loss of vehicle	E	N	N	C	Y	N
1965	Gemini 3	Incorrect lift estimate causes short landing	Landed 84 km short, crew manually compensated, decreasing short landing error	E	N	Y	C	N	Y
1965	Gemini 5	Data error of earth rotation lands Gemini 5 short	Landed 130 km short	E	N	N	D	N	N
1968	Apollo 8	Memory Inadvertently Erased	Close Call fixed manually	E	N	N	O	N	N

Year	System	Title	Result	Erroneous or Silent	Reboot Recoverable?	Missing Code?	Error Location	Computer Science?	Unknown-unknown?
1969	Apollo 10	Switch Misconfigured as bad input data to abort guidance	Vehicle tumbled, close call, recovered manually	E	N	N	D	N	N
1981	STS-1	Failure of computers to sync	Launch Scrub of First Shuttle flight	S	Y	Y	C	N	N
1982	Viking-1	Erroneous Command caused loss of comm	End of mission	E	N	N	O	N	N
1985-87	Therac-25	Radiation Therapy machine output lethal doses, user input speed	Four deaths, two chronic injured	E	N	N	C	Y	N
1988	Phobos-1	Erroneous unchecked uplinked command lost vehicle	Loss of vehicle/Mission	E	N	N	O	N	N
1988	Soyuz TM-5	Wrong code executed to perform de-orbit burn	Extra day in orbit, New code uplinked	E	N	N	C	N	N
1991	Aries - Red Tigress I	Bad command causes guidance error	Loss of Vehicle	E	N	N	S	N	N
1991	Patriot Missile	Patriot failed target intercept due to 24-bit rounding error growth in time over time	Failed to intercept scud missile, resulting in American barracks being struck, 28 soldiers killed, 100 injured	E	Y	N	C	Y	N
1992	F-22 Raptor	Software failed to compensate for pilot-induced oscillation in presence of lag	Loss of test vehicle	E	N	Y	S	N	Y
1994	Clementine Lunar Mission	Erroneous thruster firing exhausted propellant, cancelling asteroid flyby	Failed mission objective	E	N	N	C	N	N
1994	Pegasus XL STEP-1	Booster loss of control due to lateral instability	Loss of vehicle/Mission	E	N	Y	C	N	Y
1994	Pegasus HAPS	Navigation software error prematurely shut down upper stage	Unintended/low orbit	E	N	Y	C	N	N
1995	Solar and Heliospheric Observatory (SOHO)	Gyro Data used from unpowered sensor spins vehicle out of communication	Loss of mission during extended use	E	N	Y	C	N	N
1996	Ariane 5 Maiden Flight	Unprotected overflow in floating-point to integer conversion disrupted inertial navigation system	Loss of Vehicle	E	N	N	C	Y	N
1997	Pathfinder	Software priority inversion caused images to stall	Close Call for Mission Loss	E	N	N	C	Y	N
1998	Delta III	Unanticipated 4Hz Oscillation in control system led to vehicle loss	Loss of vehicle	E	N	Y	C	N	Y
1999	Mars Polar Lander	Premature shut down of landing engine due to misinterpretation of landing signature	Loss of Vehicle/mission	E	N	Y	S	N	N
1999	Mars Climate Orbiter	Metric vs. imperial units error	Loss of vehicle/mission	E	N	N	D	N	N

Year	System	Title	Result	Erroneous or Silent	Reboot Recoverable?	Missing Code?	Error Location	Computer Science?	Unknown-unknown?
1999	Titan IV B Centaur	Programming error omitting decimal in data file caused loss of control	Unintended orbit, Milstar Satellite lost 10 days after launch	E	N	N	D	Y	N
2000	Zenit 3SL	Ground software error failed to close valve.	Loss of Vehicle	E	N	N	C	N	N
2001	Pegasus XL/HyperX Launch Vehicle / X-43A	Airframe failure due to inaccurate analytical models	Loss of vehicle/mission	E	N	Y	C	N	Y
2001	STS-108 through 110	Shuttle main engine controller mix-ratio software coefficient sign-flip error	Significant close call, SME underperformance, though not extreme enough to not reach orbit.	E	N	N	D	N	N
2003	Multidata Systems Radiation Machine	Radiation Therapy machine output lethal doses, counterclockwise user input	Many injured, 15 people dead.	E	N	N	C	N	N
2003	Soyuz - TMA-1	Undefined yaw value triggered Ballistic reentry	landed 400 km short	E	N	N	C	N	N
2003	North American Electric Power Grid	Real-time software errors contribute to Widespread power outage	Widespread Loss of Power Service (2 hr - 4 days)	S	Y	N	C	Y	N
2004	Spirit Mars Exploration Rover	Repeated computer resets due to saturated memory usage.	Temporary Loss of Communication	S	N	N	D	Y	N
2005	CryoSat-1	Missing command causes loss of vehicle	Loss of Vehicle	E	N	Y	C	N	N
2005	DART (Demonstration of Autonomous Rendezvous Technology)	Navigation software errors fail mission objectives.	Loss of mission objectives	E	N	N	C	N	N
2006	Mars Global Surveyor (MGS)	Erroneous command led to pointing error and power/vehicle loss	Premature Loss of vehicle	E	N	N	C	N	N
2007	F22 First Deployment	International Date Line crossing crashed computer systems	Loss of navigation & communication	S	N	Y	C	N	N
2008	STS-124	All 4 shuttle computers fail / disagree during fueling	Fueling stopped	E	N	Y	S	N	N
2008	Quantas Flight 72, Airbus A330-303	Sensor Input spikes caused autopilot to pitch-down, resulting in crew and passenger injuries	One crew member and 11 passengers suffered serious injuries	E	N	Y	S	N	Y
2008	B-2 Spirit - Guam crash	Miscalculation in flight computers with missing input data calculated uncommanded pitch up	Crew members successfully ejected.	E	N	Y	S	N	Y
2012	Red Wings Flight 9268 TU-204 crash	Unanticipated landing circumstances coupled with	5 of 8 crewmembers killed	E	N	Y	C	N	Y

Year	System	Title	Result	Erroneous or Silent	Reboot Recoverable?	Missing Code?	Error Location	Computer Science?	Unknown-unknown?
		design features resulted in crash landing							
2015	Airbus A400M test flight	Missing software parameters during installation cause crash	Four fatalities	E	N	N	D	N	N
2015	SpaceX CRS-7	"Open Chute" command invalidated after launch vehicle failure	Possibly could have saved Dragon capsule from crash landing.	E	N	Y	C	N	N
2016	Hitomi X-ray space telescope	Error in computing spacecraft orientation led to spacecraft loss	Lost of vehicle	E	N	N	C	N	N
2017	SpaceX CRS-10	Erroneous relative state vector transmitted to Dragon	ISS rendezvous delay	E	N	N	D	N	N
2018, 2019	Boeing 737 MAX	Unanticipated software response to faulty sensor input	346 people died on two flights	E	N	Y	S	N	Y
2019	Boeing Orbital Flight Test (OFT)	Incorrect MET causes no ISS rendezvous and short mission, and uncovers other latent LOM software errors.	Failed ISS rendezvous, multi-year program delay	E	N	N	C	N	N
2019	Beresheet	Reboots cause engine shutdown on lunar descent	Loss of vehicle	S	N	N	C	N	N
2019	Chandrayaan-2 Vicram Lunar Lander	Unexpected velocity behavior during descent caused crash landing	Loss of vehicle	E	N	Y	C	N	N
2020	Amazon Web Service (AWS) Kinesis	Maximum threads reached caused cascading server outage	Loss of service, revenues.	S	N	Y	C	Y	N
2020	BD Alaris™ Infusion Pump	Infusion delivery system software causes injury/death	55 injuries, 1 death	E	N	N	C	Y	N
2021	Global Facebook Outage	Bad command causes global Facebook and cascading communication outages.	Disrupted communication, loss of revenues	S	Y	N	O	N	N
2021	ISS	Uncontrolled ISS attitude spin from erroneous thruster firing software	Close Call	E	N	N	C	N	N
2022	CAPSTONE	Bad Command causes Temporary Comm Loss	Delayed Trajectory Course Maneuver Objective, Close Call for LOM	E	N	N	O	N	N
2023	NOTAM – Notice To Air Mission	Corrupted database file causes flight cancellations	Loss of Service	S	N	Y	D	N	N
2023	ispace Hakuto-R	Invalidated Altitude data during Lunar descent loses Lander	Loss of Mission	E	N	Y	S	N	N
2023	Launcher Orbiter SN3 space tug	Uncontrolled attitude spin lost power and spacecraft	Loss of Mission	E	N	Y	C	N	N
2023	Voyager-2	Bad command causes [Temporary] Loss of communications	[Temporary] Loss of Communications	E	N	N	O	N	N

5. Results

Erroneous vs. Fail-Silent

Using the data from Table 3, Figure 10 shows the number of incidents and percent of erroneous versus fail silent manifestations. Erroneous output was over five times as likely, 85% of the cases as opposed to 15% of the cases failing silent. Critical systems should take the substantially greater likelihood of erroneous behavior into account when considering and designing for fault tolerance. Based on this, the system's operation should be evaluated with the following question in mind, "What would the impact be if the software behaved unexpectedly at this moment?" Depending on the answer, appropriate monitoring, override, and/or backup systems should be employed.

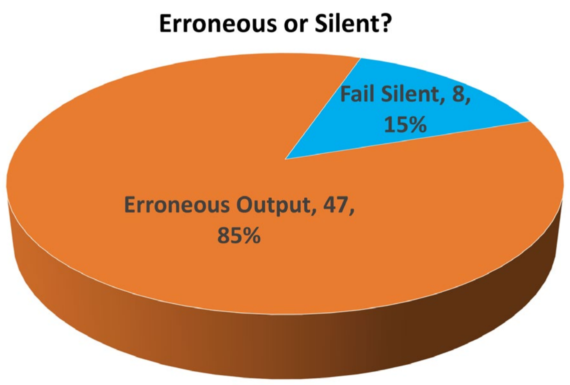

Figure 1. Erroneous vs Fail-Silent Software Manifestations

Reboot Recoverability

Figure 2 shows the subjective reboot recoverability likelihood comparing erroneous output cases and fail-silent cases. Shown here, 98% of the erroneous output cases were deemed not correctable by reboot, with only 2%, the single erroneous output case for the Patriot Missile, recoverable by reboot. Reboot recoverability seems ineffective for erroneous output cases. Fail-silent cases showed a greater chance of reboot recoverability over a small data set of 8 cases with three of eight, or 37% deemed recoverable. This implies that reboot may not be a reliabale strategy to clear fail-silent situations. Perhaps depending upon criticality, an alternate backup mitigation approach besides rebooting should be considered. Overall, reboot only was deemed effective for 4 out of 55 incidents, independent of manifestation, or about 7% of the cases.

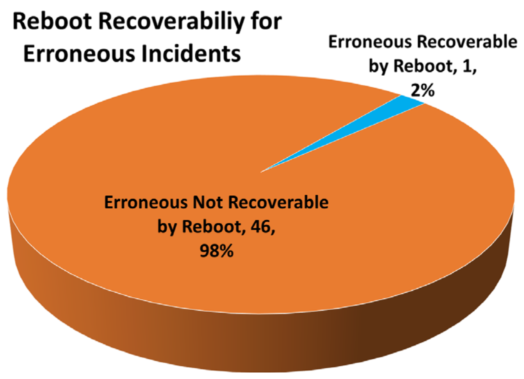

Figure 2. Reboot Recoverability for Erroneous Incidents

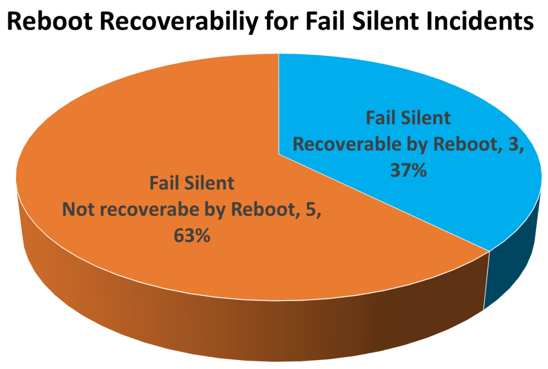

Figure 3. Reboot Recoverability for Fail-Silent Incidents

Absence of Code

Figure 4 indicates that an interestingly large 40% of these incidents were the result of the absence of code, as opposed to other causes, albeit in hindsight. The absence of code satisfies the question, "Could/should software have been added to correct this incident?", and is subjective, but includes cases such as missing requirements, incomplete understanding or modeling of the real world, unexpected inputs, and unknown-unknown subsequently discussed. This result poses an interesting concern about testing code only against requirements, and only the code that exists. If 40% of errors are in code that isn't there, how can missing code be exposed? This result should influence software requirements and testing. For example, a proportionate amount of requirements verification and unit testing should be performed on the code that exists, but a percentage of the testing should also be reserved for off-nominal cases and unexpected input scenarios, possibly exposing some of the code that is lacking.

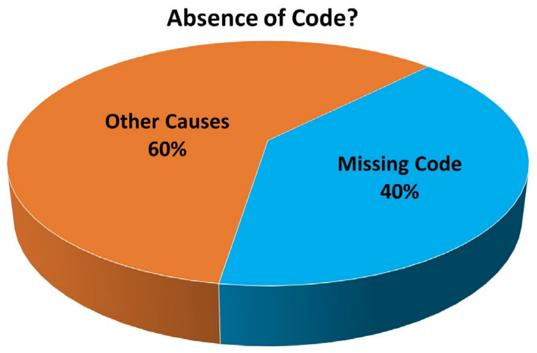

Figure 4. Absence of Code Percentages

Error Location

Figure 5 categorizes the location or point of origin for the error within the software architecture. These categories were chosen because mitigating errors between these categories is normally done with different methods, testing, processes, and procedures. Unsurprisingly, most of these errors were found to be within the code and logic itself since this category includes missing requirements, lack of response to unexpected behavior, faulty programming, and "unknown-unknowns", discussed in Section II.F, below. It also largely includes the "Sensor Input" category. Uncovering missing code during earlier phases such as unit testing or requirements verification may be a challenge, but missing code could possibly be exposed during integration testing, hardware-in-the-loop-testing, and especially with off-nominal scenario testing. Code/logic errors could also be exposed through more detailed requirements and interface control documents, focused peer reviews and comprehensive unit testing. Misconfigured data caused 16% of these errors. To combat data misconfiguration errors, special testing should be performed to assure that configurable data is validated prior to flight and reviewed by system experts, even if the software itself does not change. Unexpected sensor input accounted for 15% of all errors. While handling input could also be considered part of coding/logic, it is useful to break this out knowing that comprehensive and off-nominal input testing could be employed to uncover errors in this part of the code. Randomized input could be computer-generated to assure robustness to unexpected input. For sensor data, actual sensor hardware should be used to "test like you fly" rather than simulating sensor input. For command input errors, operational procedures should be put in place and safeguards followed to validate commands prior to issue. Command input can be considered operator error, however, safeguards such as two-stage commanding and dialog with the operator on consequences of commands could be added to software.

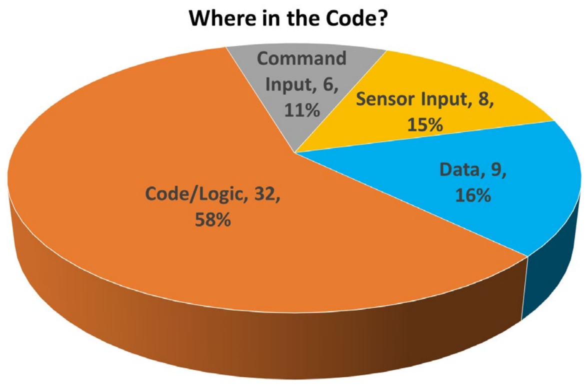

Figure 5. Error Location, Point of Origin

Computer Science/Poor Programming

Figure 6 shows the relative percent of incidents that were categorized as computer science or programming in nature as described in Section 3. Eighteen percent (18%) of the errors were subjectively considered to be more localized to within the realm of computer science. Interestingly, none of the incidents studied were the result of operating system, compiler, development tool, or programming language selection. This result, coupled with the "absence of code" result imply that effective dissimilar redundancy for fault tolerance should better employ independent requirements and test over dissimilar software platform.

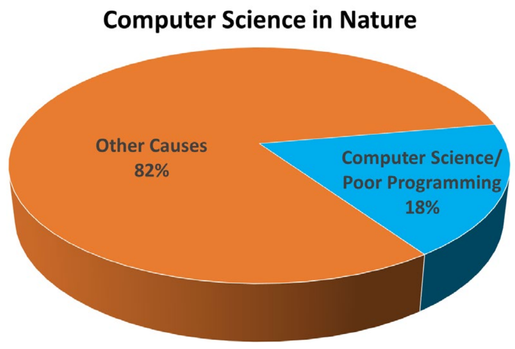

Figure 6. Computer Science or Poor Programming Related

Unknown-unknowns

Characterizing unknown-unknowns is highly subjective and can be controversial. However, the term is commonly used in aerospace practice as one justification for fault tolerance. It could be argued that given enough time and resources, each of these incidents could have been known a priori, so a subjective reasonability test was considered against each incident to distinguish "should or could this have been known within reasonable project constraints" versus, "the project did everything they should have, yet an unknown situation led to unexpected software behavior". Unknown-unknowns include cases of unknown aerodynamics after modeling, highly unusual sensor behavior, or behavior in the presence of unlikely fault situations. Given this subjectivity, the percent of these incidents that could be considered "unknown-unknown" is conservatively 16%. If one-sixth of software errors are due to things reasonably unknowable, this alone could give credence to the consideration for erroneous software backup strategies in safety-critical applications. Overall strategies to mitigate the risk of software failing during operations due to unknown-unknowns or other software failures are usually time-criticality dependent, but generally include manual human-in-the-loop control, employing dissimilar backup systems, run-time monitoring and response systems, computer reboot, entering a safe mode, or time-permitting, software reload.

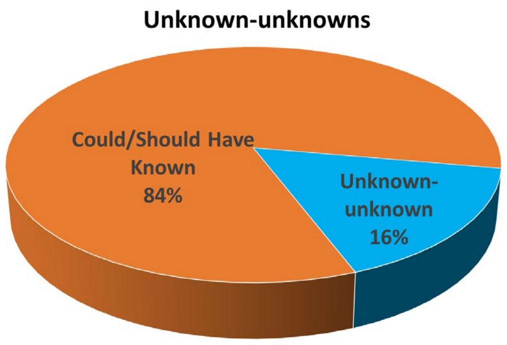

Figure 7. Unknown-unknowns

6. Conclusions

This paper enumerated a dataset of aerospace incidents involving software[1,2] since the advent of computerized automation. It analyzed aerospace failures through the eyes of the software and automation discipline to characterize and predict trends in software behavior (and misbehavior) as a design and test aid to current and future aerospace systems. It characterized how software is most likely to fail – erroneously or silent – and determined that automation predominantly fails erroneously, much more often than simply "crashing" or ceasing to output. Systems should recognize this relative risk and design accordingly. Rebooting software, though used prevalently, was evaluated to be largely ineffective to clear software failures, effective in 7% of the total cases, 37% for failing silent, and less than 2% effective for the erroneous-output case, so relying on reboot to recover from software faults should be carefully considered. This paper explored software errors relating to the absence of code as well as the prevalence of unknown-unknowns, both of which were substantial constituents in the dataset, 40% and 16% respectively. Software testing should be planned to uncover missing code through off-nominal input and integrated testing "as you fly", and backup systems should be considered to mitigate the risk of unanticipated situations and "unknown-unknowns" in safety-critical systems. A categorization determining the point of error introduction to the software architecture (code, data, sensor input, or command input) was provided to better influence processes and testing related to those areas during both development and operations. Finally, it was determined that an arguably small number of cases, 18%, were attributable to traditional computer science issues or poor programming, with none of those issues a result of operating system, programming language, or development tools. To summarize, these results indicate that erroneous and unexpected behavior, as opposed to crashing, must be considered a primary software risk, that rebooting is not reliable, and that many software errors are due to unknown or unexpected situations rather than from simple programming mistakes or development tools. Additionally, unique strategies between code, data, sensor, and command input sources should be undertaken to minimize software errors at the point of origin.

The dataset presented here is rich for further study, especially in the areas of backup systems, relationship to common-cause, and manual control for safety-critical systems. Some key questions such as, "Was this a multi-string common-cause failure?", "Was a manual or automated backup system used?", "Would a backup system have helped?", "If so, what kind of a backup system could have helped?" could be explored. Would a human-in-the-loop, a dissimilar backup, a monitor system, a focused backup, or no backup at all be the best option for each situation? Are there any trends to common backups that would have saved a class of these errors?

Other key questions could be, "What was the root cause of this error?" Looking at how these errors might have been avoided altogether has great merit. Since the software performed exactly as programmed in these cases, exploring "why" it was programmed the way it was in terms of root-cause may be a lesson to organizations producing software. "In what phase of the project could/should have this incident been discovered and averted?" is another interesting question. How much testing and what type of testing would have provided the most "bang for the buck" in averting these errors? All of these questions would be useful follow-on work against this, hopefully stagnant, dataset.

Future Trend Speculation

We provide a final speculative note regarding the use and effect of more modern software development strategies employed by new or evolving companies entering the aerospace industry. It is acknowledged that software development tools and practices such as continuous integration have enabled increased productivity and may help ensure higher quality software, however, the author believes that the rate of software/automation growth [11] has offset these practice improvements. As a response to increased volume, software development efforts have had to become more data driven and more configurable -- it simply cannot be rewritten for every configuration or for every flight. It is speculated that errors introduced through configuration data or version management will become more significant with modern software designs, though the overall occurrence of software error incidents will likely continue.

Acknowledgements

This work was sponsored and funded by the National Aeronautics and Space Administration (NASA) Engineering and Safety Center (NESC). It represents the work of several individuals on assessment teams working to improve human spaceflight safety by reducing software errors. I would like to sincerely thank each of those members who contributed to this dataset and information, in alphabetical order: Jon Berndt, Tim Brady, Jesse Couch, Tim Crumbley, Neil Dennehy, Jenny DeVasher, Captain Victor "Ike" Glover, E. Bruce Jackson, Dr. Mary Kaiser, Kylene Kramer, John LaNeave, Laura Maynard-Nelson, Dr. Paul Miner, Mike Peacock, David Root, Manuel Rosso-Llopart, Jeremy Shidner, Scott Tashakkor, John West, Daniel Williams, and the late Aron Wolf.

References

[1] Prokop, Lorraine, E., "Software Error Incident Categorizations in Aerospace", NASA Technical Publication, NASA/TP–20230012154. August 2023.

[2] Prokop, Lorraine, E., "Software Error Incident Categorizations in Aerospace", [Manuscript in Publication], Journal of Aerospace Information Systems.

[3] Harland, D., and Lorenz, R., "Space Systems Failures: Disasters and Rescues of Satellites, Rockets and Space Probes", Library of Congress Control Number: 2005922815, ISBN 0-387-21519-0, Springer Berlin Heidelberg New York, 2005.

[4] Brilliant, Susan S. John C. Knight, and Nancy G. Leveson. "Analysis of faults in an N-version software experiment." IEEE Transactions on software engineering 16.2 (1990): 238-247, 1990.

[5] Lutz, R., "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems" ISBN:0-8186-3120-1, DOI: 10.1109/ISRE.1993.324825, 1993 Proceedings of the IEEE International Symposium on Requirements Engineering, Page(s):126 – 133, 1993.

[6] NPR 7150.2D, NASA Software Engineering Requirements, NASA, March 8, 2022, URL: https://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7150&s=2D [retrieved 23 Aug 2023].

[7] NPR 8705.2C, Human-Rating Requirements for Space Systems, NASA, 10 July 2017. URL: https://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=8705&s=2C [retrieved 23 Aug 2023].

[8] RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification. RTCA, Inc., 2011.

[9] Driscoll, K., Hall, B., Sivencrona, Phil Zumsteg, P., "Byzantine Fault Tolerance, from Theory to Reality," Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23-26, 2003.

[10] Zak, D., "'Nothing ever ends': Sorting through Rumsfeld's knowns and unknowns", The Washington Post, July 1, 2021.

[11] Judas, P. and Prokop, L., "A Historical Compilation of Software Metrics with Applicability to NASA's Orion Spacecraft Flight Software Sizing", DOI:10.1007/s11334-011-0142-7, Innovations in Systems and Software Engineering, 1 Sept 2011.

Biography

Lorraine Prokop *received her BS, MS, and PhD in Computer Science from the University of Houston System. She currently serves as the NASA Technical Fellow for Software and has been with NASA for over 35 years. Her focus has been in developing and managing large real-time safety-critical software projects. She works to promote sound software architecture, software reuse, value-added software engineering process, and to ensure the safety of human-rated flight software. She is dedicated to advancing the state of the software discipline at NASA while reducing software risk and maximizing productivity for current and future spaceflight programs.*