Software Engineering for Space Exploration

Robyn Lutz

Iowa State University and Jet Propulsion Laboratory/California Institute of Technology

Abstract

Innovations in robotic spacecraft are transforming the way we explore space. Spacecraft are becoming more software-intensive in order to support smarter remote control, more productive science gathering and analysis, better troubleshooting during flight and accelerated understanding of our universe. Advances in software engineering play a vital role in space exploration by providing tools, techniques and a systematic approach to develop and service the many different kinds of software needed for robust, deep-space discovery.

Keywords

D.2.2 Design Tools and Techniques, D.2.4.g Reliability, D.2.7 Distribution, Maintenance and Enhancement, D.2.15 Software and System Safety.

1. Software engineering plays a vital role in space exploration

Innovations in robotic spacecraft are transforming the way we explore space. Spacecraft are becoming more software-intensive in order to support smarter remote control, more productive science gathering and analysis, better troubleshooting during flight and accelerated understanding of our universe. Spacecraft launched this year by NASA will map the moon's gravitational field (GRAIL), study how Jupiter and other giant planets form (Juno), and land a robotic rover to investigate the Martian environment (Mars Science Laboratory and its rover, Curiosity).

Despite our best efforts, exploration by robotic spacecraft is perilous. Spacecraft operate in imperfectly understood environments in near-Earth and deep space. Some environmental factors are hostile to software systems. For example, radiation-caused bit flips may result in software failure or unexpected software behavior. While some surprises that the spacecraft encounter are good (for example, Voyager 2 unexpectedly witnessed a rare supernova), other unexpected events or circumstances can cripple or destroy systems.

Software engineering offers some techniques that improve the odds that the spacecraft will be able to survive, contributing to its resilience to new environmental challenges and its adaptability to updates of mission requirements. We describe below some challenges involved in building and sustaining spacecraft over long missions and the contributions that software engineering has made in facing those challenges successfully.

A variety of software supports the scientific discovery that is space exploration. The software for space exploration includes the flight software that is onboard planetary orbiter and lander spacecraft, the navigation and hazard-avoidance software that drives Martian rovers, the operational software used on Earth to monitor and control the missions, and the software used to analyze the scientific data returned by

the missions. Software navigates space probes to other planets, acquires scientific data from rovers and instruments, analyzes the acquired data onboard and transmits the data back to Earth for further processing, helps scientists visualize and interpret the results, and applies the newly discovered knowledge in spinoff solutions to other problems on Earth.

Software engineering plays a vital role by providing tools, techniques and a systematic approach to develop and service the many different kinds of software needed for space exploration. Advances in software engineering such as model-based software development, formal verification and product line development help ensure that the software built meets each exploration mission's needs and operates each spacecraft safely and correctly.

In this article we focus on software engineering for robotic spacecraft. These are unmanned vehicles controlled remotely by software commands sent from Earth or encoded in programs onboard the spacecraft. Robotic spacecraft are the forward guard in our efforts to explore space since they can go farther than manned missions, building knowledge of places beyond our current capacity to send humans. Because many have to travel far and survive a long time to achieve their mission, they must be exceedingly robust. Their software is highly fault tolerant to detect and respond to anomalies that occur during operations. Their software also can be updated in flight to adapt to changes in the mission's needs or to accommodate the failure of components.

While we here discuss primarily spacecraft developed or managed by NASA's Jet Propulsion Laboratory (JPL), software for robotic spacecraft is also produced by other NASA facilities, international space agencies (see the sidebar "International space exploration"), commercial facilities and research institutions. Many spacecraft have parts and software from multiple companies and agencies, and multi-national collaboration is a hallmark of space exploration. For example, the 2016 ExoMars Trace Gas Orbiter mission is a joint European Space Agency/NASA-JPL mission that will study sources of methane and possibly other gases on Mars.

> Sidebar 1: International space exploration. Space exploration is an international endeavor. Many missions are international with several geographically distributed space agencies, companies and research institutions cooperating to achieve what no single one could. Limited budgets, different areas of expertise, the internationalization of science, sharing of risks, and opportunities for cooperating networks of spacecraft will encourage future international collaboration in space. See http://en.wikipedia.org/wiki/List_of_space_agencies for more information.

2. Software engineering issues in space exploration

We describe five software engineering issues met in developing and sustaining robotic spacecraft. These illustrate some difficulties encountered in this domain and software engineering solutions used for them. The solutions appear likely to have use in other systems facing similar requirements for safety or mission criticality, robustness, remote operation, complexity, or autonomy.

2.1 Designing for Change

Change is typical over the lifetime of a spacecraft's mission. Software systems must be designed to accommodate it, especially on long-lived spacecraft traveling to distant planets. For example, the New Horizons spacecraft to study Pluto and its moons launched in 2006 but will not reach Pluto until 2015. Software engineering for spacecraft thus devotes considerable effort to trying to anticipate and plan for change. Some changes will involve updates to requirements. Others will involve updates to onboard software technology, such as improved image compression algorithms or additional autonomous

reconfiguration options. The spacecraft must also, as is discussed below, be robust enough to maintain its essential functional capabilities despite failures.

Spacecraft software can be, and is, updated during flight and operations. Bugs are fixed, adaptations to changed context are uploaded, and enhancements to what the software can do or how well it can do it are made. On missions that travel for long periods to reach their destination, it is normal to develop the software needed for later surface operations or orbit during the lengthy cruise phase. Software change and augmentation after launch is part of the plan. It allows the software to be sequentially updated and customized to the next phase of the mission. The sidebar "Deep space: the farthest human-made objects" describes instances in which the Voyager ground and flight software needed and received new algorithms.

Sidebar 2: "Deep space: the farthest human-made objects" The two Voyager spacecraft, launched in 1977, are now the farthest human-made objects from Earth. They continue to return invaluable data as they approach the heliopause. Recently, Voyager 2 discovered a strong magnetic field that holds the interstellar cloud together [http://voyager.jpl.nasa.gov/]. To get that far, Voyager 2 ground and flight software have had to be updated during flight. Soon after launch, Voyager 2's primary receiver failed and its backup receiver was reduced to "hearing" in a very narrow, changing frequency band. Design practices, such as those described in section 2.1, allowed the team to respond quickly to the problem. A new algorithm was designed and implemented, so that prior to sending any software commands to the spacecraft, ground operations could tune the transmission to the receiver's current state. Flight software was updated when a new algorithm was required to obtain camera images at Neptune. Without it, the low sunlight levels, combined with the torque imparted when the tape recorder was turned on and off, would have caused images to be smeared. The new feature automatically fired the attitude jets to compensate for the spacecraft torque at the longer exposure rate, allowing stunning images of Neptune to be acquired and transmitted to Earth (Fig. 1).

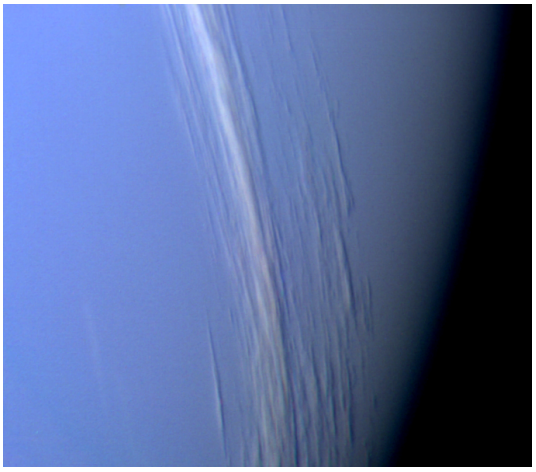

Figure 1. Neptune Clouds, taken by Voyager 2 spacecraft two hours before closest approach. Courtesy NASA/JPL-Caltech

The capability to update the software during operations is essential. In a study of 199 post-launch anomaly reports on seven spacecraft, where the anomalies were classified as critical and software-related by the projects, about 7% resulted in a new software requirement either to handle a rare but high consequence event (e.g., failure of both redundant units) or to compensate for hardware failures or limitations [6]. Such requirements changes seem to be typical of systems in which the hardware is

inaccessible (such as spacecraft) or difficult to access (such as implanted medical devices), or in systems in which the operating environment is dangerous (e.g., high-radiation, high-temperature or hostile). In such systems, when hardware fails or degrades, software is often updated to compensate for the failure.

Many spacecraft and rovers have excellent hardware and software reliability, allowing them to far exceed their planned mission lifetimes. For example, the twin Mars Exploration Rovers, Spirit and Opportunity, finished their original 90-day primary mission in 2004, but 90 months later, Opportunity still continues its trek across Mars.

The software architecture for the spacecraft is thus designed to accommodate changes to the software. Given the turnover in personnel that can occur over the lifetime of a spacecraft and the likelihood that changed circumstances will result in changes to the software requirements, it is important to keep available the reasoning behind the design decisions that were made. Retaining the rationales in the models, documentation and operational flight rules helps prevent the introduction of errors when software is updated.

2.2 Fault tolerance

Fault tolerance is a specialized kind of planning for change. It is not surprising, given the long life and hostile environments encountered by spacecraft, that failures occur. Developing robust software that can handle failures and other conditions that threaten the mission begins with a thorough systems and software hazards analysis. Software requirements to protect the spacecraft from mission-critical failures are derived from these hazards analyses.

An example of successful fault tolerance occurred when the Cassini spacecraft, currently on an extended mission to Saturn, unexpectedly ceased communication with ground operations on Nov. 2, 2010. It was later determined that this was caused by a flipped bit. The software had been designed to configure the spacecraft to a safe but degraded state when communication was lost. As described at the Cassini website, the onboard fault-protection software quickly switched to a backup computer and shut off non-essential power loads. It switched to an alternate (low-gain) antenna, pointing it toward the sun to improve the chances of communicating with operators. Communication was restored within the hour, at which point human operators began a slow and careful check of the spacecraft followed by recovery activities such as clearing error log files and turning science instruments back on. Recovery actions were completed on November 24, and the Cassini spacecraft continues to be healthy at the time of this writing.

The fault protection software on spacecraft continually monitors for deviations between the expected spacecraft state and the actual spacecraft state. Fault recovery software is especially important because it is usually invoked when something has already gone wrong on the spacecraft. As spacecraft and their missions become more complicated, the number of unwanted scenarios that the software must handle increases. Software is growing to handle not just failures (e.g., the Cassini bit flip), but also a range of other unexpected situations (e.g., novel usage scenarios) and contingencies (e.g., debris temporarily blinding the spacecraft camera). However, as Dvorak et al. note, extending the techniques of onboard fault protection to cover a wider range of software failures can also increase algorithmic complexity, which makes the software harder to verify [2].

2.3 Remote control

To accommodate the long distances between spacecraft and the ground operators on Earth that fly them, the spacecraft must be given some autonomy. For example, the roundtrip light time to send a radio signal command to Cassini and receive acknowledgement that the command was received correctly is over two and a half hours. Many decisions and activities must be accomplished much faster than this, forcing

operators on earth to permit the remote robotic spacecraft's software to provide limited real-time control.

For example, when the Mars Exploration Rover spacecraft landed in 2004, onboard software used radar, image and sensor data to track and respond to wind velocity as the spacecraft landed on Mars. A subsequent report by Johnson et al., described the performance benefit the autonomous software provided: "Had DIMES [the Descent Image Motion Estimation System] not been available to measure the steady state velocity, the EDL [Entry-Descent-Landing] system would not have fired TIRS [the retro rockets] and the total velocity would have been just on the threshold of airbag performance. Furthermore, the velocity would have been to the East toward the rockier terrain" [4].

Rovers on Mars, for which the roundtrip light time is currently about 40 minutes, routinely drive themselves short distances. The rover on the Mars Science Laboratory spacecraft that will be launched in late 2011 has autonomous navigation software that allows it to plan and drive a path to an interesting feature as far as 50 meters away while avoiding any hazards in its way. Figure 2 shows the growth of the rovers: from left to right, one of the 2004 Mars Exploration Rovers (Spirit and Opportunity), the 1997 Mars Pathfinder rover, and the 2011 Mars Science Laboratory rover, Curiosity.

Figure 2. Rover comparison. Courtesy: NASA/JPL-Caltech

A success story regarding autonomy on spacecraft is the Automated Sciencecraft Experiment (ASE), described by Chien in [1]. This software, which was a co-winner of the NASA 2005 Software of the Year, has flown on the Earth Observing satellite EO-1 for several years. The ASE software makes decisions onboard about which science observations are most important to make and which science data is important to downlink to Earth. For example, when the Erebus Volcano in Antarctica erupted in 2004, the software detected it, and quickly planned additional data collection of the event without waiting for human intervention. The effect was that the volcano was re-imaged within about six hours from initial detection. Without the ASE software self-direction, the response time would have been much slower, from 12 to 26 days.

The ASE autonomy also was estimated to reduce the cost of planning and mission operations by over a million dollars per year. Software engineering techniques cited as important to its high reliability were redundant, layered architecture and supporting models, a rigorous systems and requirements engineering process, multi-level testing process, and an incremental spiral development [1]. The benefits of autonomous software were demonstrated again in 2006, when the Nyamuragira Volcano in the eastern Democratic Republic of the Congo erupted near a populated area. The EO-1 satellite, acting as part of a

volcano sensor web, collected precise data regarding the vent location and expected direction of the lava flow to assist with planning on the ground, as described at http://earthobservatory.nasa.gov/Features/VolcanoSensorWeb/.

2.4 Grappling with growth

A simple system is easier to understand, and hence easier to build correctly, to test, and to update as needed over time. However, as seen above, much of the promise of improved knowledge discovery will depend on more and better data, and on smarter spacecraft. Making spacecraft smarter, in terms of adaptability (to respond to new mission requirements), fault tolerance (to isolate and recover quickly from faults), and autonomy (to make decisions locally on-board rather than wait for human operators), pushes us toward bigger and more complicated software.

The growth of software in size and complexity is ongoing. As reported by Dvorak et al. [2], and shown in Table 1, the trends for human and robotic NASA missions from 1968 through 2005, measured in non-commented source lines of code, show an exponential growth rate of a factor of 10 approximately every 10 years. One of the key challenges of software engineering for spacecraft is how to handle this growth in size and complexity. Especially for a spacecraft, where once launched, there is no way to pull it back into the shop for repair, smarter but more fragile software would be a bad trade.

An example of how easy it is to make a critical design flaw and how hard it can be to fix it in flight is the mission-critical "Sol 18" anomaly that occurred on the Spirit Mars Exploration Rover (MER) on Jan. 21, 2004. An error in an off-the-shelf file services module allowed incorrect dynamic memory allocation during operations. The impact was then compounded by two configuration errors. As a result, out-of-memory events caused repeated processor resets, interrupted communication with operators on Earth, and caused battery power depletion that threatened the spacecraft's mission.

A protective feature in the software design called "crippled mode" was essential to the rover's eventual recovery. This contingency mode had limited but essential capabilities that enabled the team eventually to prevent continuous resets and, after two weeks of intense team effort, to restore the health of the

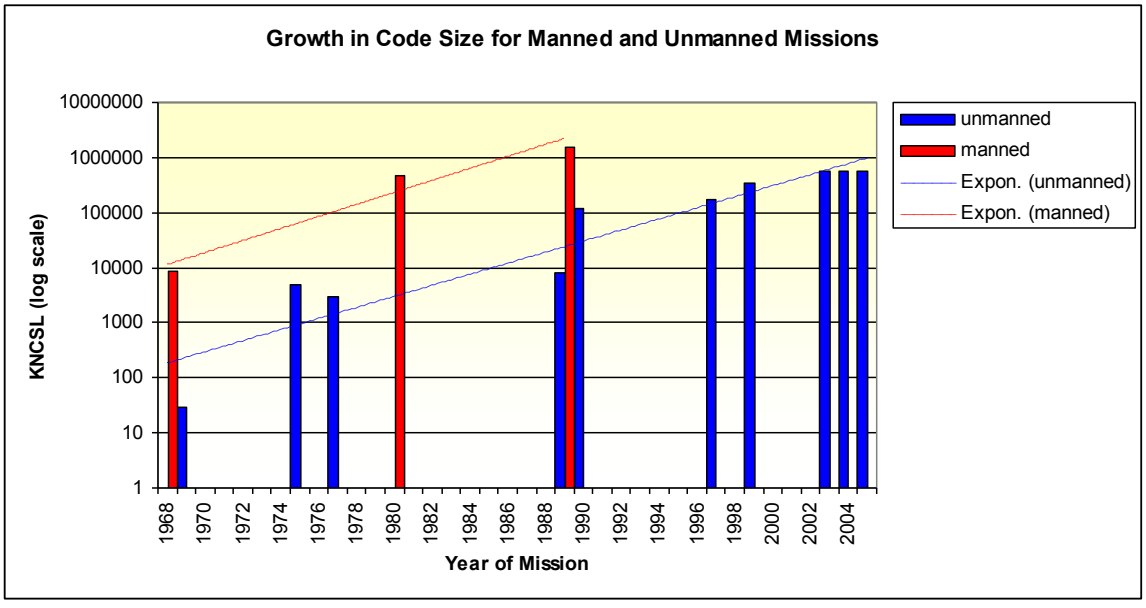

Table 1. Grappling with growth [Source: G. Holzmann, as reported in [2]]

spacecraft. An account of the anomaly by Reeves and Neilson concludes that, "without this capability, the MER mission may well have been lost. The crippled mode mechanism allowed us to regain control of the vehicle." [9]. This incident showed the importance of assuring compaction for deleted files in the directory structures for long duration missions, the difficulty of understanding and managing all the interactions, the risk that the unexpected behavior of the Commercial-Off-The-Shelf software introduced, the importance of verifying the operational assumptions, the need for telemetry that reveals the current software state, the impracticality of operational tests long enough to reveal this behavior (i.e., over 11 sols), and the important role of automated test data analysis.

2.5 Software engineering in embedded systems

Software engineering goes hand-in-hand with system engineering on robotic spacecraft. The software is part of a larger system whose devices it monitors and controls. Software engineering is part of the integrated infrastructure that makes spacecraft exploration work.

Experience with the Galileo spacecraft exemplifies how any software solution must take into account broader system and environment interactions. The High Gain Antenna (HGA) on the spacecraft, which had been folded up like an umbrella during launch, failed to unfurl when commanded to open in 1991. During the four years before arrival at Jupiter, extraordinary efforts were made to try to open the antenna. Without the HGA, only a small percentage of the science data that would be collected at Jupiter could be transmitted to Earth.

The solution involved what Nilsen and Jansma termed the "first complete reload of flight software ever performed on a deep-space mission" [7]. But recovery from the HGA anomaly required more than just a software solution. The workarounds "were truly a team effort involving a system approach that included science, flight, ground, hardware, and software". High-rate data from the Jupiter encounter would need to be buffered and then slowly trickled to Earth over the Low Gain Antenna (LGA).

To accomplish this, new software was designed and uploaded to Galileo to allow data processing and advanced data compression onboard. The new onboard software alone meant that the spacecraft could send ten times the number of images and data possible otherwise possible without the HGA. Additional changes were made on the ground to increase the data capacity. Hardware and software improvements to the deep space network with which the spacecraft communicated with Earth included advanced antenna arraying, low-noise receivers and improved modulation schemes. These improvements also greatly increased the amount of science data that could be received from Jupiter.

Organizationally, the teams worked together to prioritize mission and science goals and to develop new plans for the Jovian encounter that would maximize mission return. While the data returned could not equal the original plan, the effective data downlink was increased more than 400 times, from 10 bps (the LGA rate) to about 4.5 Kbps, and most of the mission's goals were achieved.

3. What works

Many of the software engineering techniques (e.g., hazard analysis, requirements-based testing, formal verification) used for other safety-critical systems such as airplanes, cars, pacemakers, power plants and factory-floor robots have been adapted for space exploration. The flow of ideas is a two-way street, since many of these techniques have been extended for space exploration and the spinoffs returned to the marketplace.

Areas in which robotic spacecraft share many of the same concerns and have provided software engineering solutions for other, similar systems are model-based diagnostics and recovery, design for

reuse, risk management, delay-tolerant network software (which preserves loss-less communication even if communication with the spacecraft is slow or interrupted), distributed sensor network technologies, redundancy management, product line engineering, and resource-constrained (power, memory, mass, bandwidth) systems.

The Laboratory for Reliable Software at JPL introduces technologies to improve software practice on spacecraft and other mission software. Examples include model checking to verify critical components and interfaces such as the Mars Science Laboratory spacecraft's flash file system for flight, static source code analyzers as standard practice, and institutional coding standards including steps to reduce risk related to multi-threaded software. Many of these advances are also advisable for other critical domains, as well as for improved software productivity in non-critical domains. Holzmann describes in [3] how, in order to support the code review process on the Mars Science Laboratory project, they developed the SCRUB tool. It combines results from multiple source code analyzers with results from code walkthroughs to make the review process more efficient and the code more reliable. It has been used on the project for three years for all reviews of both manually written code and code auto-generated from high-level formats.

The long operational lifetimes of many spacecraft inevitably involves personnel turnover. This means that the documentation, preservation and maintenance of knowledge about the spacecraft are especially important for such systems. We have proposed to adopt software product-line engineering techniques for use in single long-lived, robotic spacecraft to help identify potential alternatives, document assumptions, and make decisions regarding alternatives and changes, based on cost and value, both during development and operations [7]. Anticipated change can be usefully modeled as variability, such that making an anticipated change after launch is handled as if it were making different choices for a new product.

The design of spacecraft missions balances highly innovative technologies which promise more and better science data, with an aversion to risk driven by often unique mission opportunities. For example, only once every 175 years do the planets Jupiter, Saturn, Uranus and Neptune line up so that a single spacecraft can fly past all of them, as Voyager 2 did.

More broadly, the software engineering experience gained and techniques developed for mission-critical robotic spacecraft transfer readily to other safety-critical systems. We have learned that rare events that threaten the spacecraft do occur and must be planned for, that static analysis and requirements-based testing significantly reduce operational defects, that overly strict requirements unnecessarily add complexity and consequent risk, that traceability to requirements and rationale forestalls injection of design defects, that it is important to design not just for failures but also for undesirable contingencies, that restraint in the introduction of complexity reduces risk, that requirements always change after launch, that partial autonomy enables science gains not possible otherwise because of long communication times, that time invested in a well-thought-out architectural design pays off in increased sustainability, that there is no substitute for expertise and the power of a committed team (see Fig. 3), and that the loss of knowledge over the lifetime of a long-lived system can be fought with model-based engineering and good traceability.

Figure 3. Mars Reconnaissance Orbiter Mission Team Members Celebrate Orbit Insertion Success. Courtesy: NASA/JPL-Caltech

One challenge that will be essential to address is the analysis, visualization and management of very large amounts of science data. The Mars Reconnaissance Orbiter, which went into orbit in 2006, gathered 100 terabits of data in its first four years at Mars. This was more than three times the data gathered up to then by all other deep-space robotic spacecraft combined. Another area of great interest is the software engineering of multiple, distributed, formation-flying spacecraft, perhaps with some having specialized science instruments that can cooperate to learn about space in ways not previously possible.

4. Conclusion

The good news for students is that as the capabilities of robotic spacecraft increase, the need for software engineering for space exploration will continue to grow. As Carl Sagan famously said, "Exploration is in our nature. We began as wanderers, and we are wanderers still. We have lingered long enough on the shores of the cosmic ocean. We are ready at last to set sail for the stars" [10]. Or, quoting a recent article on the interplanetary program, "If you can't have a good time coming to work and building robots to send to Mars, give it up, man" [5].

Acknowledgments

Thanks to Margaret Smith, Martin Feather and Scott Morgan for additional examples.

References

[1] S. Chien, "Integrated AI in Space: The Autonomous Sciencecraft on Earth Observing One," *Proc. 21st Nat'l Conf Artificial Intelligence and 18th Innovative Applications of Artificial Intelligence Conf*, AAAI Press, 2006.
[2] Daniel L. Dvorak, ed. *Final Report, NASA Study on Flight Software Complexity*, 3 Mar 2009; http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf.
[3] G. J. Holzmann, "SCRUB: a tool for code reviews," *Innovations in Systems and Software Engineering*, vol. 6, no. 4, 2010, pp.11-318.
[4] A. E. Johnson, R. G. Willson, Y. Cheng, J. Goguen, C. Leger, M. Sanmartin, L. Matthies, "Design Through Operation of an Image-Based Velocity Estimation System for Mars Landing," *Int'l J Computer Vision*, vol. 74, no. 3, 2007, pp. 319-341.
[5] J. Kluger, "Scientific Illiteracy After the Shuttle: Are America's Smartest Days Behind Her?," *Time,* 11 Jul 2011; http://www.time.com/time/health/article/0,8599,2082213,00.html.

[6] R. Lutz and I. C. Mikulski, "Empirical Analysis of Safety-Critical Anomalies During Operations," *IEEE Trans. Software Eng*, vol. 30, no. 3, 2004, pp. 172-180.
[7] R. Lutz, D. M. Weiss, S. Krishnan and J. Yang, "Software Product Line Engineering for Long-Lived, Sustainable Systems," *14th Int'l Software Product Line Conf (SPLC)*, 2010, pp. 430-434.
[8] E. N. Nilsen and P. A. Jansma, "Galileo's Rocky Road to Jupiter," *NASA ASK Magazine*, vol. 42, Spring, 2011.
[9] G. Reeves and T. Neilson, "The Mars rover spirit FLASH anomaly," *Aerospace Conf*, IEEE Press, 2005, pp. 4186–4199.
[10] Carl Sagan, *Cosmos*, Random House, 1980.

Biographical sketch

Robyn Lutz is a professor of computer science at Iowa State University and in the software system engineering group at Jet Propulsion Laboratory/California Institute of Technology. Her research interests include software engineering of safety-critical systems, product lines, and the specification and verification of requirements for autonomous systems. Lutz received a PhD in Spanish literature from the University of Kansas. She is a senior member of the IEEE. Contact her at rlutz@iastate.edu.

Contact information:

Robyn Lutz
Dept. of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, IA 50011
515-294-3654 (phone)
515-294-0258 (fax)
rlutz@cs.iastate.edu