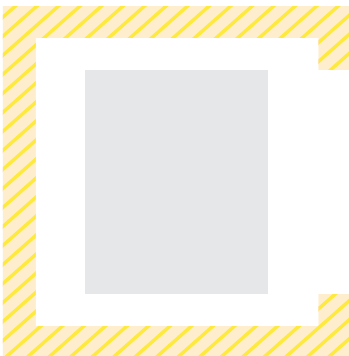




Landing a Spacecraft on Mars

Gerard J. Holzmann

How much software does it take to land a spacecraft safely on Mars, and how do you make all that code reliable? In this column, Gerard Holzmann describes the software development process that was followed. —*Michiel van Genuchten and Les Hatton*



THOUSANDS OF PEOPLE worked on design, construction, and testing of the hardware for NASA's latest mission to Mars. This hardware includes not just the rover itself with its science instruments, but also the cruise stage, which guided the *Curiosity* rover to Mars, and the descent stage, with the intricate sky-crane mechanism that gently lowered the rover to the surface on 5 August 2012.

All this painstakingly developed, tested, and retested hardware is controlled by software. This software was written by a relatively small team of about 35 developers at NASA's Jet Propulsion Laboratory (JPL). Obviously, the control software is critically important to the mission's success, with any failure potentially leading to the loss of the spacecraft—as well as to headline news around the world.

The control software onboard the spacecraft consists of about 3 MLOC. Most of this code is written in C, with a small portion (mostly for surface navigation) in C++. The code executes on a radiation hardened CPU. The CPU is a version of an IBM PowerPC 750, called RAD750, which is designed for use in space. It has 4 Gbytes of flash memory, 128 Mbytes of RAM, and runs at a clock-speed of 133 MHz.

About 75 percent of the code is auto-generated from other formalisms, such as state-machine descriptions and XML files. The remainder was handwritten specifically for this mission, in many cases building on heritage code from earlier Mars missions.

The *Curiosity* Rover is the seventh spacecraft that NASA has successfully landed on Mars. Previous spacecraft include

- two *Viking* landers in 1976,
- the *Pathfinder* minirover in 1996,
- the two Mars Exploration Rovers *Opportunity* and *Spirit* in 2004, and
- the *Phoenix* Mars lander, which was launched in 2007 but reused the design of the failed Mars Surveyor lander from 2001.

Each new mission is more complex and uses more control software than its predecessor. But that's putting it mildly. As in many other industries, code size is growing exponentially fast: each new mission to Mars uses more control software than all missions before it combined:

- the *Viking* landers had about 5 KLOC onboard,
- *Pathfinder* had 150 KLOC,

COMPOUND ANNUAL GROWTH RATE IN MARS MISSIONS' CODE



The Compound Annual Growth Rate, as described in previous columns,¹ in flight software for spacecraft over the last 36 years comes out at roughly 1.20—close to the median value of 1.16. The Mars Science Laboratory software is comparable to other safety-critical systems that were described in this column, such as the Tokyo Railway system,² Honeywell's Flight Management System,³ and Airbus.⁴ What sets this system apart is that it must operate reliably at a distance of millions of miles from Earth, making it inaccessible to standard types of maintenance and repair.

References

1. M. Genuchten and L. Hatton, "Compound Average Growth Rate for Software," *IEEE Software*, vol. 29, no. 4, 2011, pp. 19–21.
2. D. Avery, "The Evolution of Flight Management Systems," *IEEE Software*, vol. 28, no. 1, 2011, pp. 11–13.
3. S. Burger, O. Hummel, and M. Heinisch, "Airbus Cabin Software," *IEEE Software*, vol. 30, no. 1, 2013, pp. 21–25.
4. K. Tomita and K. Ito, "Software in an Evolving Train Traffic Control System," *IEEE Software*, vol. 28, no. 2, 2011, pp. 19–21.

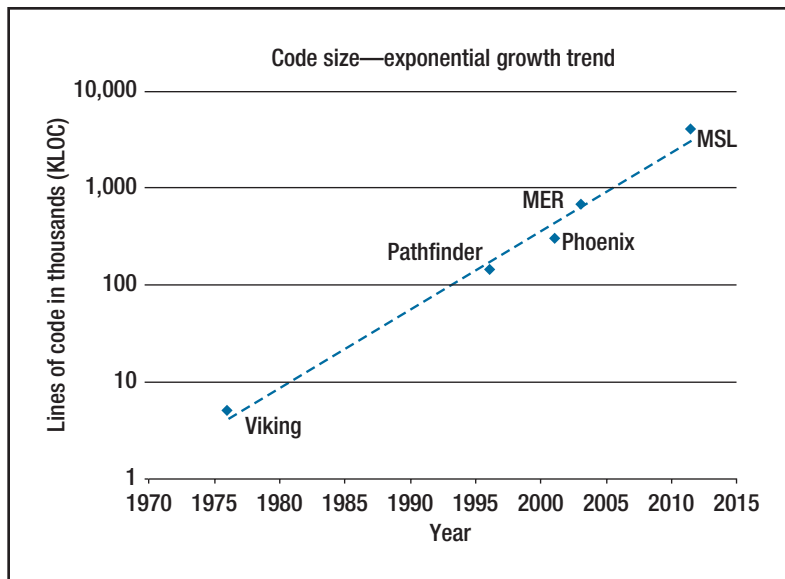


FIGURE A. The amount of flight code that is flown to land spacecraft on Mars has grown exponentially in the last 36 years. Its Compound Annual Growth Rate comes out at roughly 1.20—close to the median value of 1.16 from previous columns.

fully managed process. The three main control points in this process are prevention, detection, and containment.

Prevention

The best way to make software reliable is to prevent the introduction of defects from the start. We tried to do this in a number of ways.

First, we adopted a strong new coding standard for the mission (which was later also adopted as a common standard for all software development at JPL¹). Although most software development projects use coding standards, we followed a somewhat different approach in the definition of this one. To define the rules in this standard, we first looked at everything that had gone wrong in previous space missions. We categorized the problems that could be attributed to software and devised a small set of rules that could prevent these classes of errors. Next, we focused on those rules for which we could mechanically check compliance—for example, with a static analyzer. Our coding standard captured those rules, and only those rules. Therefore, the rules in this coding standard cannot be silently ignored (as is often done with other standards). We mechanically checked compliance with all the rules on every build of the software. Any deviations were reported and became part of the input to the downstream code review process.

Second, we introduced a flight software developer certification course, focused in part on software risk and defensive coding techniques. Every software developer is required to complete this course and pass the exams before they can touch flight software. The course covers the coding standard's rationale, as well as general background on computer science principles and the basic structure of spacecraft control software. Some of this material is also presented to more senior managers at

- the *Phoenix* lander had 300 KLOC,
- the Mars Exploration Rovers each had 650 KLOC, and
- the MSLRover upped the ante to 3 MLOC.

There's clearly no single magic tool or technique that can be used to secure the reliability of any large and complex software application; rather, it takes good tools, workmanship, and a care-

JPL to secure a common knowledge base regarding the challenges of mission-critical software development (although in the latter case, the material is presented without the pressure of an exam at the end).

Detection

The next best thing to preventing defects is to detect them as early as possible in the software development cycle. To do this, we adopted a range of state-of-the-art static source code analyzers, paired with a new tool-based code review process.²

The challenges in conducting peer code reviews on millions of lines of code are well known.³ The process we adopted therefore shifted much of the burden of the routine checks (such as checks for common types of coding errors, compliance with the coding standard, or risky code patterns) to background tools. A complete integration build of all MSL flight software was performed nightly, with all checkers running over the code in parallel to the builds. We jointly used four different static analyzers with close to a hundred simpler custom-written checking scripts that verified compliance with various types of requirements that are harder to encode in static analyzers (such as rules against the use of tabs in code or rules for the types of header files that must or must not be used).

We used the static analyzers Coverity (www.coverity.com), Codesonar (www.grammatech.com), Uno,⁴ and, toward the end of the software development, the newer tool Semmler (<http://semmler.com>). Each of these tools has different strengths and tends to find different types of flaws in the code; there's surprisingly little overlap in the output. The results of the nightly analyses were made available in the single uniform interface of the Scrub tool,² which also integrated peer comments collected during the code review phase for each

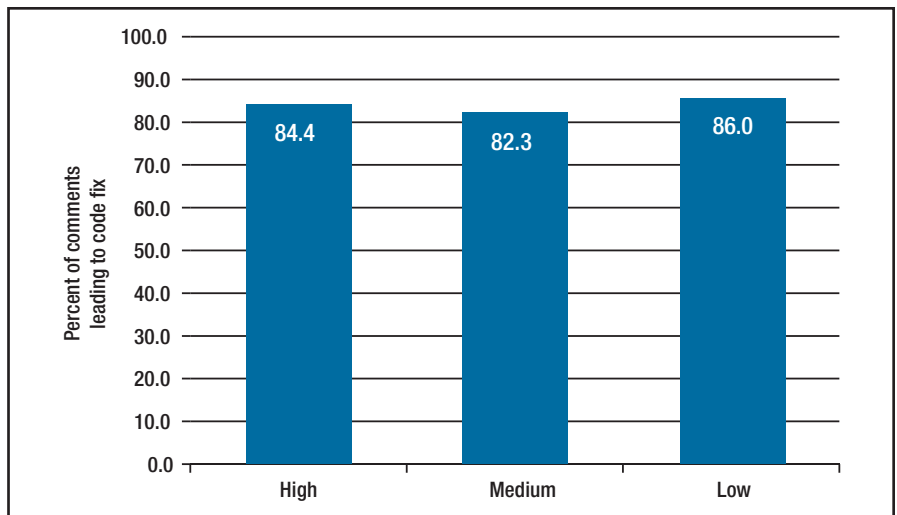


FIGURE 1. The percent of peer comments resulting in a code fix in the Mars Science Laboratory code review process between 2008 and 2012, by priority, showing that all comments were taken equally seriously. The majority of comments led to changes in the code.

module. What was perhaps different about the code review process we followed was that we did all peer reviews offline rather than in face-to-face meetings. We required the module owner to respond to all reports (generated by tools or peers) with a simple *agree*, *disagree*, or *discuss* response. *Agree* meant that the module owner agreed with the finding and committed to changing the code. *Disagree* indicated a difference of opinion, where the module owner believed that the code was correct as previously written and shouldn't be changed. *Discuss* meant that the report was unclear and the owner needed more information before they could determine a fix or no-fix resolution.

At the end of the offline reviewing period for each module, we held a single face-to-face meeting with reviewers, module owner, and the flight software lead present to discuss any remaining disagreements. In almost two hundred peer code review sessions held for the MSL mission, approximately 80 percent of all peer comments and tool reports were accepted with an immediate *agree* from the module owner.

Only the remaining 20 percent of the reports or tool warnings therefore required discussion in the review meetings, leading to a final resolution of either *fix* (which in some cases overruled an earlier *disagree* response from the module owner), or *no fix* (see Figure 1). In all, roughly 10,000 peer comments on the code were processed in this way, together with approximately 25,000 tool reports. As shown, the vast majority of these peer comments and tool reports led to changes in the flight code to either address an issue or to prevent a tool warning from recurring in later builds.

We added another layer of checking with the application logic model-checking techniques (<http://spinroot.com>) to analyze key parts of the multithreaded code for possible race conditions or deadlocks. We analyzed five critical subsystems of the control software's in this way. In one case our findings led to a complete redesign of the subsystem to prevent the types of problems that the model checker had uncovered.

Apart from the static analysis, peer code reviews, and logic model-checking

analyses, every software module had to pass rigorous unit and integration tests, both in a desktop software-only context with simulated hardware and on the real hardware in flight system testbeds. The requirement for the unit tests was, as is common in this type of application, to realize full code coverage. Note, though, that this requirement isn't always easy to comply with, especially when using defensive coding techniques. After all, defensive coding techniques often deal with cases that should be impossible under nominal circumstances but that protect against the consequences of as-yet unknown and unpredictable types of errors. Defensive code, then, can sometimes be flagged as unreachable, which would violate a coding rule that bans the inclusion of unreachable code. The solution in this case is to devise demonic

tests that can trigger the defensive code and show the circumstances under which it could be executed.

Containment

The final layer of defense in a reliable software system is more structural in nature. It's the mechanism that targets defects that weren't prevented or detected earlier, to make sure that these residual defects do not spread beyond the module in which they occur and bring down the system as a whole. One method to achieve this is to provide redundant backups. The principle of redundancy is well understood in the hardware design and relatively easy to implement in those cases. We can, for instance, put a second set of thrusters on a spacecraft, or a second transmitter and receiver. The assumption here is that hardware failures in duplicated equipment are mostly uncorrelated.

This same principle is much harder to follow in software. Obviously, running the same software on multiple CPUs doesn't protect against software failures (a lesson that has been learned a few times too many in the past⁵).

The principle of containment was most prominently used in software that was used to control the critical landing sequence, which is often described as the "seven minutes of terror." The hardware for the spacecraft was designed to be "dual string," which means that many critical components are duplicated for reliability. This duplication includes the CPU; there are two completely separate CPU and memory subsystems.

The backup CPU is designed to take over control of the spacecraft if the main CPU fails. However, if the software caused the failure, it clearly wouldn't help much for the second CPU to execute precisely the same code after such a switch. During the landing sequence, the backup CPU therefore executed a simplified, stripped-down

version of the landing software. That software version was called "Second-Chance" (although, within the software team it was also referred to as "Last-Chance"). As we know now, the main CPU succeeded in guiding the spacecraft to a flawless landing on Mars, and this part of the system wasn't needed. To date, no significant anomalies have revealed themselves in the flight software. ☺

Acknowledgments

The research described in this article was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. *JPL Coding Standard for Flight Software Written in the C Programming Language*, Jet Propulsion Laboratory California Institute of Technology, 2009; http://lars-lab.jpl.nasa.gov/jpl_coding_standard_c.pdf.
2. G.J. Holzmann, "Scrub: A Tool for Code Reviews," *Innovations in System and Software Eng.*, vol. 6, no. 4, 2010, pp. 311–318.
3. G.W. Russell, "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, Jan. 1991, pp. 25–31.
4. G.J. Holzmann, "Static Source Code Checking for User-Defined Properties," *Proc. 6th World Conf. Integrated Design and Process Technology (IDPT 02)*, 2002; <http://spinroot.com/uno>.
5. J.L. Lions, *ARIANE 5: Flight 501 Failure*, tech. report, Centre National d'Etudes Spatiales, 1996; www.di.unito.it/~damiani/ariane5rep.html.

GERARD J. HOLZMANN is a senior research scientist and fellow at NASA's Jet Propulsion Laboratory at the California Institute of Technology. He was a member of the Mars Science Laboratory flight software team. Contact him at gholzmann@acm.org.

IEEE STC 2013

25th IEEE Software Technology Conference


8-11 April 2013

Salt Lake City, Utah, USA

Register today!

<http://www.sstc-online.org/>



 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.