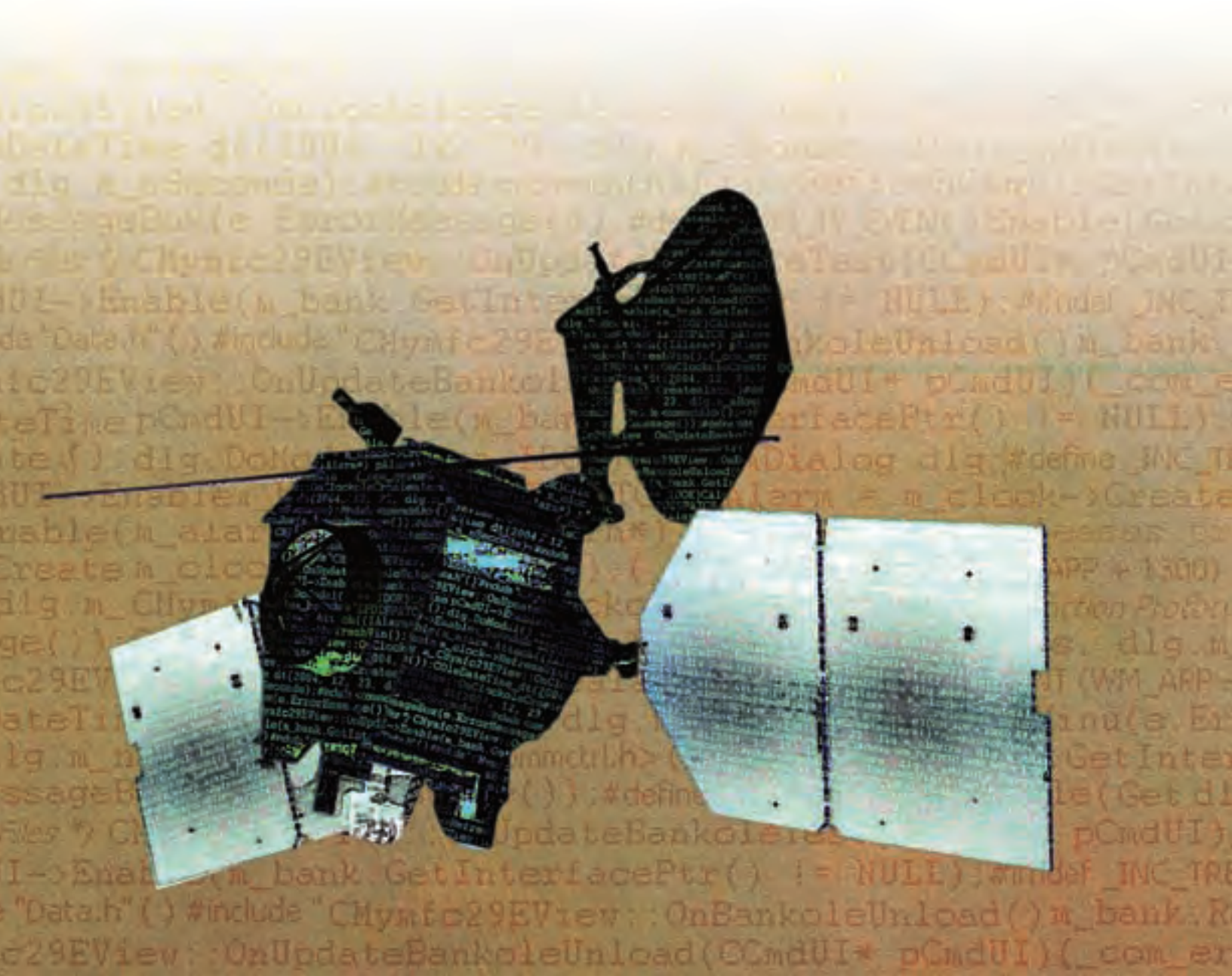
Is Software Broken?

BY STEVE JOLLY

A few years ago my attitude toward the design and development of space systems fundamentally changed. I participated in a Kaizen event (part of Lockheed Martin's Six Sigma/lean culture) to ascertain contributing factors and root causes of various software overruns and schedule delays that can precipitate cost and schedule problems on both large and small space programs alike, and to propose process improvements to address those causes. I didn't anticipate that software's modern role in spacecraft development could itself be a problem.



A Kaizen event is a tool used to improve processes, a technique popularized by the Toyota Corporation that is now used widely in industry. Stakeholders and process experts come together to analyze or map an existing process, make improvements (like eliminating work that adds no value), and achieve commitment from both the process owners and the users. In this particular Kaizen, software and systems engineering subject-matter experts came together from across our corporation to participate. We had data from several recent spacecraft developments that we could study.

We all suspected some of the cause would be laid at the feet of systems engineering and program management, with the balance of the issues being inadequate adherence to established software development processes or processes that needed improvement. But the Kaizen event is designed to ensure we systemically addressed all the facts, and our large room soon became a jungle of flipcharts covered in a dazzling array of colored sticky notes, each chart representing a different aspect of the problem and each sticky note a potential root cause. The biggest problem was there seemed to be somewhere around 130 root causes.

What we had expected based on other Kaizen events was that this huge number of root causes was really a symptom of perhaps a half-dozen underlying *true* root causes. A small number can be addressed; we can form action plans and attack them. Hundreds of causes cannot be handled. Something was wrong either with the Kaizen approach or with our data.

I came away from the event somewhat puzzled. We resumed the activity several months later, but we did not materially improve upon our initial list and get to a satisfying short list. However, several of us began to notice a pattern. Even though we couldn't definitively link the large majority of causes, we found that problems in requirements issues, development, testing, and validation and verification of the actual code all revolved around interfaces. When viewed from a higher altitude, the preponderance of the causes collectively involved all the spacecraft subsystems. With such systemic coverage of functions on the spacecraft, it was tempting to conclude that software processes were broken. What else could explain what we were seeing?

I couldn't accept that conclusion, however. I knew many of our software engineers personally—had walked many developments with them—and although we had instances of needing better process adherence and revised processes, something else was clearly at work. I reflected on eight recent deep-space missions that spanned the mid-nineties through 2008, and it became clear that software has become the last refuge for fixing problems that crop up during development. That fact is not profound in itself; what is profound is that software is actually able to solve so many problems, across the entire spacecraft.

For example, while testing the Command and Data Handling (C&DH) subsystem on the Mars Reconnaissance Orbiter, we discovered a strange case of hardware failure deep in a Field-Programmable Gate Array (FPGA) that would result in stale sun-sensor data and a potential loss of power, which could lead to mission failure. To make the interplanetary launch window, there was no time to change or fix the avionics hardware. Instead, we developed additional fault-protection software that was able to interrogate certain FPGA data and precipitate a reset or "side swap" should the failure occur. Indeed, software is usually the only thing that can be fixed in assembly, test, and launch operations and the only viable alternative for flight operations. In fact, close inspection during our Kaizen showed that most of our 130 root causes could be traced to inadequate understanding of the requirements and design of a function or interface, not coding errors. Suddenly the pieces began to fall into place: it's all about the interfaces. Today, software touches everything in modern spacecraft development. Why does software fix hardware problems? Because it can. But there is a flip side.

In the past software could still be viewed as a bounded subsystem—that is, a subset of the spacecraft with few interfaces to the rest of the system. In today's spacecraft there is virtually no part of the system that software does not have an interface with or directly control. This is especially true when considering that firmware is also, in a sense, software. Software (along with avionics) has become the system.

This wasn't the case in the past. For example, Apollo had very few computers and, because of the available technology, very limited computing power. The Gemini flight computer and the later Apollo Guidance Computer (AGC) were limited to 13,000–36,000 words of storage lines.[1] The AGC's interaction with other subsystems was limited to those necessary to carry out its guidance function. Astronauts provided input to the AGC via a keypad interface; other subsystems onboard were

controlled manually, by ground command, or both combined with analog electrical devices. If we created a similar diagram of the Orion subsystems, it would reveal that flight software has interfaces with eleven of fourteen subsystems—only two less than the structure itself. Apollo's original 36,000 words of assembly language have grown to one million lines of high-level code on Orion.

Perhaps comparing the state-of-the-art spacecraft design from the sixties to that of today is not fair. The advent of object-oriented code, the growth in parameterization, and the absolute explosion of the use of firmware in evermore sophisticated devices like FPGAs (now reprogrammable) and application-specific integrated circuits (ASIC) have rapidly changed the art of spacecraft design and amplified flight software to the forefront of development issues. Any resemblance of a modern spacecraft to one forty years ago is merely physical; underneath lurks a different animal, and the development challenges have changed. But what about ten years ago?

Between the Mars Global Surveyor (MGS) era of the mid-nineties to the 2005 Mars Reconnaissance Orbiter (MRO) spacecraft, code growth in logical source lines of code (SLOC) more than doubled from 113,000 logical SLOC to 250,000 logical SLOC (both MGS and MRO had similar Mars orbiter functionality). And this comparison does not include the firmware growth from MGS to MRO, which is likely to be an order of magnitude greater. From Stardust and Mars Odyssey (late 1990s and 2001) to MRO, the parameter databases necessary to make the code fly these missions grew from about 25,000 for Stardust to more than 125,000 for MRO. Mars Odyssey had a few thousand parameters that could be classified as mission critical (that is, if they were wrong the mission was lost); MRO had more than 20,000. Although we now have the advantage of being able to reuse a lot of code design for radically different missions by simply adjusting parameters, we also have the disadvantage of tracking and certifying thousands upon thousands of parameters, and millions of combinations. This is not confined to the Mars program; it is true throughout our industry.

But it doesn't stop there, and this isn't just about software. Avionics (electronics) are hand-in-glove with software. In the late 1980s and early 1990s, a spacecraft would typically have many black boxes that made up its C&DH and power subsystems. As we progressed—generation after generation of spacecraft avionics developments—we incorporated new electronics and new packaging techniques that increased the physical and functional density of the circuit card assembly. This resulted in several boxes becoming several cards in *one* box; for example, the functionality of twenty-two boxes of the MGS generation was collapsed into one box on Stardust and Mars Odyssey. Then, with the ever-increasing capability of FPGAs and ASICs and simultaneous decrease in power consumption and size, several cards became FPGAs on a *single* card. When you hold a card from a modern C&DH or power subsystem, you are likely holding many black boxes of the past. The system is now on a chip. Together with software, avionics has become the system.

BOTTOM LINE: THE GAME HAS CHANGED IN DEVELOPING SPACE SYSTEMS. SOFTWARE AND AVIONICS HAVE BECOME THE SYSTEM.

So then, there are no magic few underlying root causes for our flight software issues as we'd hoped to find at our Kaizen Event, but the hundreds of issues are unfortunately real. Most revolve around failed interface compatibility due to missing or incorrect requirements, changes on one side or other of the interface, poor documentation and communication, and late revelation of the issues. This indicates our systems engineering process needs to change because software and avionics have changed, and we must focus on *transforming* systems engineering to meet this challenge. This is not as simple as returning to best practices of the past; we need new best practices. The following are a few ways we can begin to address this transformation:

1. Software/firmware can no longer be treated as a subsystem, and systems engineering teams need a healthy amount of gifted and experienced software systems engineers and hardware systems engineers with software backgrounds.
2. We need agile yet thorough systems engineering techniques and tools to define and manage these numerous interfaces. They cannot be handled by the system specification alone or by software subsystem specification attempting horizontal integration with the other subsystems; this includes parameter assurance and management.
3. Using traditional interface control document techniques to accomplish this will likely bring a program to its knees due to the sheer overhead of such techniques (e.g., a 200-page formally adjudicated and signed-off interface control document).
4. Employing early interface validation via exchanged simulators, emulators, breadboards, and engineering development units with the subsystems and payloads is an absolute must.
5. If ignored, interface incompatibility will ultimately manifest itself during assembly, test, and launch operations and flight software changes will be the only viable means of making the launch window, creating an inevitable marching-army effect and huge cost overruns.

Bottom line: the game has changed in developing space systems. Software and avionics have become the system. One way to look at it is that structures, mechanisms, propulsion, etc., are all supporting this new system (apologies to all you mechanical types out there).

Today's avionics components that make up the C&DH and power functions are systems on a chip (many boxes of the past on a chip) and, together with the software and firmware, constitute myriad interfaces to everything on the spacecraft. To be a successful system integrator, whether on something as huge as Orion and Constellation or as small as a student-developed mission, we must engineer and understand the details of these hardware–software interfaces, down to the circuit level or deeper. I am referring to the core avionics that constitute the system, those that handle input-output, command and control, power distribution, and fault protection, not avionics components that attach to the system with few interfaces (like a star camera). If one merely procures the C&DH and power components as black boxes and does not understand their design, their failure modes, their interaction with the physical spacecraft and its environment, and how software knits the whole story together, then software will inevitably be accused of causing overruns and schedule delays. And, as leaders, we will have missed our opportunity to learn from the past and ensure mission success.

A final note of caution: while providing marvelous capability and flexibility, I think the use of modern electronics and software has actually increased the failure modes and effects that we must deal with in modern space system design. Since we can't go back to the past (and who would want to?), we must transform systems engineering, software, and avionics to meet this challenge. I am ringing the bell; we need a NASA–industry dialogue on this subject. ●

Note: Kaizen is a registered trademark of Kaizen Institute Ltd. Six Sigma is a registered trademark of Motorola, Inc.

STEVE JOLLY is with Lockheed Martin Space Systems. He has development and flight operations experience from many deep-space missions and was the chief systems engineer for the Mars Reconnaissance Orbiter. Most recently he was a member of the NASA integrated design optimization team for Orion and is currently the chief systems engineer for GOES-R.

1. James E. Tomayko, *Computers in Spaceflight: The NASA Experience*, NASA Contractor Report 182505 (Washington, DC: NASA History Office, 1988).