# NASA's Mission Reliable


**Like industrial development organizations, the US space agency struggles with the challenge of creating reliable software. NASA's deep space community is attacking its software crisis via two complementary approaches—one stressing the power of engineering discipline, the other the potential of automated code generation and verification.**

*Patrick Regan*
New Jersey Network Public Television and Radio

*Scott Hamilton*
*Computer*

Both predictable and unpredictable hazards await the spacecraft, robots, and scientific instruments that humans dispatch to explore our solar system. The toughest hazard to harden a space probe, orbiter, or rover against may be one of the most predictable: the known presence of unknown bugs in even the most rigorously tested software.

As this article goes to press, spacecraft from Europe, Japan, and the United States are converging on Mars, and some sophisticated software systems are approaching their ultimate test. With all eyes turning toward the red planet, some observers will recall the software-related failures of the Mars Polar Lander and Mars Climate Orbiter missions in 1999—and pay special attention to the performance of NASA's Mars Exploration Rovers, twin robotic science platforms about the size of a golf cart.

Meanwhile, far beyond Mars, the Cassini spacecraft cruises toward a rendezvous in March with Saturn. The Voyagers launched in the 1970s report back from the boundary region between our sun's sphere of influence and interstellar space. Orbiting observatories take the measure of the universe. And back in California, the next billion-dollar project is already in development at Jet Propulsion Laboratory, managed for NASA by Caltech. Scheduled for launch in 2009, the Mars Science Laboratory mission aims to land a rover as big as a car for a tour lasting more than a year. Many more missions in the $200 million range are in flight, in development, or on the drawing boards.

## HIGHER STAKES, GREATER RISKS

On increasingly frequent and ambitious missions, these remote laboratories radio back pictures of other worlds; measure radiation, particles, and fields in the interplanetary environment; probe the chemistry and dynamics of planetary bodies; and, above all, equip scientists to test theories, answering long-standing questions and raising new ones.

NASA and other space agencies rely on ever larger, more complex software systems to do more challenging science. Because of the nature of their missions and the distance from ground control—with communications requiring from tens of minutes to hours of "round-trip light time"—spacecraft and rover systems demand increasing autonomy that introduces new dimensions of complexity.

This trend raises the stakes and increases the risks that a problem older than the expression "software engineering" poses: It's not getting any easier to produce reliable software.

Within the earthbound labs responsible for designing and flying NASA's space missions, computer scientists and software engineers have been

0018-9162/04/$20.00 © 2004 IEEE

*Mars Exploration Rover—artist's conception. Courtesy NASA.*

mobilized as never before to attack this problem. It may be that the perpetual "software crisis," first declared at NATO's 1968 conference on software engineering, will humble NASA as it has other organizations that have tried to address it. On the other hand, the urgency and unique resources that NASA brings to the problem might produce breakthrough solutions.

Despite conspicuous and even tragic failures, the US space agency has engineered the seemingly impossible and made it look routine. If NASA can solve its own software crisis, it may help boost software engineering and commercial development to a higher stage of maturity.

## FROM GROUND CONTROL TO AUTONOMOUS SYSTEMS

Traditional mission control and planning have been done on the ground, with humans in the loop generating a time-based command sequence that tells the spacecraft what to do and when. Notwithstanding the occasional surprise, this approach has been quite successful; however, with the many small missions on the books, traditional ground operations costs have become prohibitive.

Past spacecraft systems included limited autonomy programs such as attitude control, fault protection, and orbit insertion, entry, and landing that were tried and tested over decades and hardcoded into the systems. However, new science mission requirements and more frequent missions have necessitated efforts to generalize autonomous spacecraft control across multiple missions.

Traditional spacecraft control systems contain a fault-protection subsystem that monitors the system's behavior and "safes" the spacecraft by pointing its solar panels toward the sun, reporting a possible malfunction to ground control, shutting down unnecessary systems, and waiting for instructions from Earth. Although such fault-protection mechanisms have had a long history of success, safing a spacecraft at critical junctures such as orbit insertion is obviously a recipe for disaster. Perhaps more important, scientists have come to expect exponentially more from these missions.

Safing a spacecraft or rover for minor faults that do not put it at risk and waiting hours for instructions can greatly reduce the amount of scientific data that can be obtained and sent back to Earth—indeed, such behaviors could preclude opportunistic science such as a comet fly-by.

Although it is a hostile environment for physical hardware, deep space is a relatively benign environment for mission control because extended periods of time are available to plan nonemergency maneuvers. A rover on the Martian surface, however, needs extensive autonomy to accomplish complex missions in which it may have only seconds to avoid toppling into a ditch.

Autonomous control poses some of NASA's trickiest problems and stimulates some of its most advanced research. Yet the agency has more mundane issues to deal with as well.

## SOFTWARE QUALITY INITIATIVE

During a career that began with semiconductor engineering for Mariner missions to Mars and Venus in the 1960s, Tom Gavin, the senior member of the Jet Propulsion Laboratory's flight community, has seen software rise to prominence not only as an enabling technology but also as a major risk and cost factor. Gavin has made it his mission to improve JPL's software processes, better integrate software production with systems engineering, and encourage research that focuses on a bulging portfolio of flight projects.

In the Mars Exploration Rover, entry, descent, and landing are software-controlled. Gavin says, "People see the mechanical system of this vehicle, parachutes coming out, airbags coming out, and bouncing on the ground. But that is a software-driven system, and a lot of the project risk is in how robust that software is." Software accounts for a large share of project cost too, in coding, independent verification and validation, and testing—four testbeds for the Mars Exploration Rover alone, running 24 hours a day in three shifts, with deadlines dictated by the movements of the planets.

Missing a deadline that celestial mechanics impose could trigger a long, costly delay—22 months for a mission to Mars, 19 to Venus, or 13 to Jupiter. Keeping such a deadline can exact a dif-

ferent kind of penalty. Gavin notes that the software for PathFinder, which landed successfully on Mars in 1997, was largely undocumented.

In an effort he says is still in its infancy, Gavin has charged the JPL software community with developing a kind of discipline long established—and recently codified—on the hardware side. A few years ago, engineers captured 40 years of hardware experience in three books on flight project practices, design principles, and mission assurance. Now the lab is investing $4 million a year in a software quality initiative aimed at getting developers to codify and buy into a similar set of software practices. A survey currently under way will help define the baseline for this effort and for forward-looking research.

More effective matchmaking between practitioners' needs and applicable research is an integral part of the software quality initiative. Gavin is optimistic that the initiative will succeed by involving the practitioners in documenting their practices.

## RESEARCH GOES DEEP

Meanwhile, exploratory research—at NASA facilities including Ames, Dryden, Glenn, Goddard, Johnson, Langley, and Marshall as well as JPL—aims at ensuring the reliability of software for future missions. The unique challenges of deep space have brought an infusion of funding to software reliability research at JPL, with close ties to affiliated groups at Ames. Researchers are applying advanced technology for both code synthesis and code verification of software that runs the gamut from artificial intelligence-based autonomy software—which uses methods such as rapid propositional deduction and adaptive neural nets to uncover faults or execute plans—to large systems with more conventional kinds of complexity.

JPL established a Laboratory for Reliable Software in 2003, with model-checking guru Gerard Holzmann as its founding member. Mission Data System (MDS) is a bigger enterprise within JPL's Interplanetary Network Directorate that has similar motivation.

In 1998, just as it was about to launch six independently designed missions in the span of six months, JPL became acutely aware of the need to make more effective use of its software engineering resources and to reuse software common to all missions. At that time, avionics engineer Robert Rasmussen championed MDS as a multimission architectural framework that could unify software and systems engineering, from the conception of a mission through development and flight.[1]

When NASA people use the term MDS, they could be referring to the architecture, the idea, the million lines of code that currently embody it, development of mission software for the Mars Science Laboratory, a set of development processes, or related research. In any case, MDS offers a multimission framework for building, testing, and reusing software that will fly in spacecraft, land in rovers, and operate here on Earth. This is a striking departure from established practice, in which highly compartmentalized development efforts have produced essentially one-off software systems for each space mission.

**Programming is a human effort, and people make mistakes.**

Between them, Holzmann's research and the MDS project don't begin to encompass all the resources being brought to bear on ensuring that NASA software will reliably do what needs to be done. But they do represent the range of approaches NASA is pursuing, and they illustrate a philosophical tension that both polarizes and helps to energize the whole endeavor.

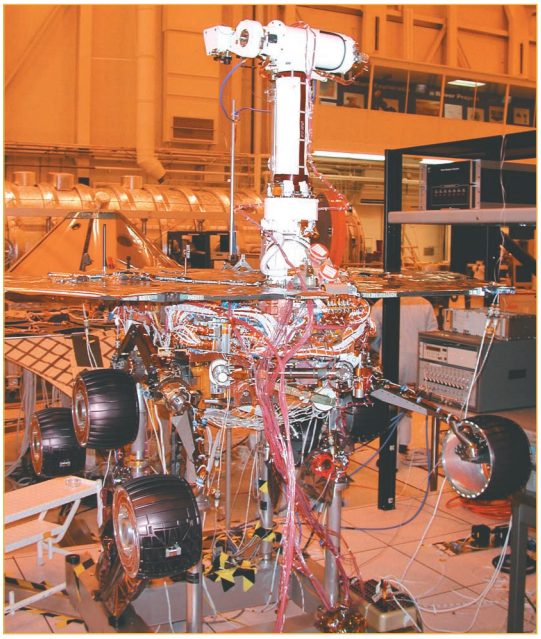
## TO ERR IS HUMAN

The basic observations underlying Holzmann's work are that programming is a human effort and that people make mistakes. He cites estimates that around 50 software defects remain in 1,000 lines of newly written uncommented code, and around 10 remain in code that's been thoroughly tested. Extreme measures of additional testing can push the number of residual defects per 1,000 lines down toward one, but that still adds up to a lot of bugs in a large system, and few products ever reach that level.

Statistics for novice and experienced programmers are essentially the same. "An experienced programmer tends to make just as many mistakes," Holzmann says, "but they are much more complex, they're subtle, they require a lot of deep thought. If he makes a mistake, it is usually much more difficult to find."

Testers are human too, and following this line of reasoning leads to the conclusion that human-driven testing processes are, like programming practices, inherently flawed. For Holzmann, this alone makes automating alternative methods an imperative. With a combination of optimism and patience, Holzmann is determined to convince his colleagues of the power of automated model checking techniques as an alternative to the conventional approach to software testing.

Methods like model checking, static analysis, and runtime analysis offer the ability to do exhaustive



***Mars Exploration Rover at the JPL spacecraft assembly facility. Courtesy NASA.***

exploration of source code, testing even the most unlikely scenarios. Conventional testing, a time-consuming operation for which the allotment of time tends to get squeezed as the delivery or launch date approaches, may root out the most likely failure scenarios and more subtle vulnerabilities; however, it can never offer the possibility of finding all the bugs.

Furthermore, testing methods lose ground all the time. "We build systems more complex than we can understand and more complex than we can check," Holzmann asserts. "Conventional testing processes, which were designed for deterministic, sequentially operating systems, haven't really changed in 30 years. Yet systems now are multithreaded, which makes them nondeterministic because of process and thread scheduling—all the interleaving is different in every run, events occur at unpredictable moments—and the test methods are simply not designed to handle that."

In contrast, he says, researchers are beginning to win the battle of software analysis. Winning, in this context, means that formerly unassailable problems show signs of yielding to improved algorithms and the steady march of processing power. By Holzmann's calculation, a small, automated test he devised in 1980 would have kept a computer busy for seven days, if he'd had one powerful enough to run it at all. In 2000, with a thousand times more memory and processing speed readily available, the same test could be done in seven seconds. Today, it would take two seconds; in two more years, it should take just one.

By the late 1970s, some believed formal methods could be automated and eventually scaled up. By the late 1980s, Holzmann had developed Spin, probably the best known and most widely used model checker in the world.[2] In the late 1990s, he and his colleagues at Bell Labs used Spin and associated technology for the first time in the development of a real product when they verified call-processing code for an Internet Protocol-based telephone switch.

Although the chunk of code they checked was relatively small—10,000 lines out of roughly 25 million—it was functionally the heart of the system and a nightmare of potentially fatal interactions between concurrent processes. This application proved a few important claims for the technology, including that it could extract models from source code automatically. The creation of a "test harness" to extract models from C code was a one-time investment, reusable throughout the development process.

JPL researchers who were using Spin to verify code for the Cassini mission drew Holzmann into NASA's orbit in the 1990s. Within weeks of his move to California, he had helped to build a test harness for verifying code already flying on the Mars Exploration Rovers and was seeking ways to contribute to the Mars Science Laboratory. At JPL, Holzmann is working to build a core group of theorists, system builders, and "energizers" that will interact with a virtual team of a few hundred to show that these techniques work.

## SPINNING IN SPACE

NASA researchers, including Ames' Klaus Havelund and Thomas Pressburger, were early and insightful experimenters with the Spin model checker. Together, they developed their own innovative verification and testing technology, Java PathFinder 1, which translated high-level source code to Promela, Spin's formal language.[3] Subsequently, for Java PathFinder 2, Ames' Willem Visser and Havelund developed a new model checker, in which a Java virtual machine interprets the source code.[4] Work is under way to integrate Java PathFinder 2's model-checking technology with Java PathExplorer, a separately developed tool for runtime analysis and monitoring.[5]

This effort began with Deep Space 1, a technology validation flight. Part of the mission called for the Remote Agent software to take control of the spacecraft for two days. Havelund and Ames' John Penix ran part of the Plan Runner, a critical Remote Agent code component, through a Spin cycle as a verification experiment. This check automatically found five bugs, concurrency errors that were corrected before flight.

Despite that check and 800 hours of preflight testing, Remote Agent hit a deadlock six hours after

it was activated. A team assembled to analyze the system's 12,000 lines of Lisp found the bug within five hours—the same kind of bug that Havelund and Penix had found in their earlier work—in a module that had been reworked after their verification experiment. Interestingly, the team opted not to fix this particular bug because the chances of its occurring again were extremely small. The bug did not recur—whereas a bug fix had a greater chance of causing problems—again, showing how rare these bugs are to begin with and why they are so hard to find with conventional testing.

That episode highlighted both model checking's potential and its current limitations. "Particularly for concurrent software where there is a lot of interleaving," says senior research scientist Michael Lowry, "model checking can be much better than testing."

According to Lowry, the increasing number of threads in some mission software is not the only trend driving up the risk of concurrency errors. Smaller numbers of complicated threads can have the same effect. Beyond that, he says, "If you're interacting with an environment in complicated ways, the whole system becomes concurrent. Think about a rover on Mars. As you move away from time-based sequencing, you have lots of different interleavings between what the environment does and what your software will do. That's exactly where you're going to get a vehicle that's capable of dealing with a rich environment—both for science return and for its own survival—and that's a situation where concurrency occurs even if the software program has only one thread."

Yet another trend highlights the challenge of scaling model checking to deal with large programs. Numbers that Lowry cites as rough guides to this trend are the 30,000 lines of code developed in the 1980s for the Cassini mission, 120,000 lines of code for the mid-1990s development of the Mars PathFinder, and 428,000 lines of code for the Mars Exploration Rover. Others suggest that the software for the Mars Science Laboratory could grow to several million lines of code.

## FINDING ALTERNATE PATHS

When Havelund and Penix began hand-translating Deep Space 1 code for Spin to check, they could analyze around 30 lines of code a day. Automating translation in Java PathFinder boosted analysis to around 1,500 lines of code per day, limited mainly by memory. At that point, however, the researchers felt constrained by Spin and Promela. That's when Visser and Havelund developed a model checker that uses a Java virtual machine. According to Visser, this model checker could "analyze all the behaviors, all the paths through the code, because we had control of scheduling, so we could schedule all the possible interleavings." Visser says that creating the first version took only three months, but they have been working for four years to perfect it.

**The software for the Mars Science Laboratory could grow to several million lines of code.**

The largest program they have analyzed so far is 8,000 lines of prototype rover software with six complicated threads, translated to Java from the original C++. Like Spin, Java PathFinder is expected to become more effective simply by riding the hardware curve and benefiting from improvements in automated translation. Penix is writing a C++ to Java translator so that Java PathFinder can be used on C++ code as well.

These researchers also aim to improve the effectiveness of automated software analysis by coming at it from another direction. Whereas Java PathFinder searches the space of paths through the code to find one that has an error in it, Java PathExplorer focuses on detecting errors in a single path. The key, Havelund says, is using more powerful temporal logic "which can, for example, reason about future and past time logic at the same time, reason about real time, and reason about values."

Java PathExplorer runs the program once, recording the system's state throughout that execution. Working with an execution trace in which all events of interest have been logged, the tool employs one set of algorithms for exact analysis to determine whether or not there is an error in that particular trace. It uses another set of algorithms to infer whether other possible permutations of the trace could contain errors.

Recent enhancements include strengthening the tool's temporal logic and extending its capability to high-level data races. An example of the latter would be a problem in which one thread updates $x$ and $y$ coordinates in a single operation, and another thread reads them separately, one at a time. Thus, if the first thread can update both coordinates between the reading of $x$ and $y$, a "correct" operation of the program could yield the wrong coordinate pair.

Havelund and Visser plan to use the underlying Java PathFinder and Java PathExplorer technology to generate tests automatically. They also have combined these model checking and runtime analysis tools in a testing environment for K9, a planetary rover.

*Mars Science Laboratory—artist's conception. Courtesy NASA.*

## COMPONENTS IN CONTEXT

In a complementary effort at Ames directed at improving model checking scalability, Dimitra Giannakopoulou and Corina Pasareanu focus on modular or compositional verification. Their approach decomposes the analysis of a program into analysis of its components that might be easier to check than the whole, and it does this in a way that guarantees the properties of the reconstructed whole. These researchers also are experimenting with the K9 rover software.

Given an abstract model of a component's behavior and a required property, Giannakopoulou and Pasareanu have developed techniques for formulating assumptions, which encode the contexts or environments in which the component will satisfy the property. When the component's environment is available, it is checked against the assumption to ensure correctness of the whole. "We split the component behavior into behavior that is controllable and uncontrollable by the environment," says Giannakopoulou, "and get as a result how controllable behavior affects the satisfaction of required properties."

These researchers have developed a framework based on the use of learning algorithms and model checking for formulating assumptions incrementally. When it's not clear what the component's environment will be, the fallback is a twist on runtime analysis—generate a monitor based on the assumption, instrument the program, monitor the environment against the assumption during deployment, and trigger recovery code when the assumption is violated.

Because a lack of design information blocks this approach at the model level, Giannakopoulou and Pasareanu are extending their approach to work directly at the software level. Where design models of a system are not available, they want to apply the same techniques to code—generating assumptions and performing the same kind of "modular reasoning" on the source code. It's an aspiration at this point, but they have some ideas about how to make it work and the collaborations needed to test their ideas.

## DYNAMICS OF STATIC ANALYSIS

Source code is in fact the starting point for static analysis, another kind of tool being honed on NASA software. Though formal, this method differs sharply from model checking, and it produces different kinds of results. According to Guillaume Brat, an Ames-based researcher, static analysis can analyze code on the basis of program semantics without constructing a model or running the program. Brat and his collaborator, Arnaud Venet, have been working to optimize static analysis for NASA and to prove its practical potential in software coding, unit testing, and integration.

Static analysis does not help much with concurrency, deadlocks, and data races. It can, however, do a thorough job of rooting out errors in programming style—such as uninitialized variables and pointers, out-of-bounds array access, and invalid arithmetic operations—that could corrupt data or even crash a system at runtime. Almost by definition, such errors are common and plentiful, but static analysis has the potential to weed them out.

This type of analysis produces a kind of mathematical hologram—described as a computable approximation to the set of values arising dynamically at runtime when executing a program—that researchers can view from various angles and probe for details. The method's scientific basis is abstract interpretation, which expresses a program in terms of equations, and lattice theory, which offers techniques for solving the equations. The output essentially is three sets of results: Errors the tool is sure are errors, code that is surely correct, and apparent problems that might be errors or might be something else, such as dead code.

The objective is to compute numerical invariance at every program point. The analysis must guess what values an integer can take for any execution of the program. For every concrete operation in the program, such as addition, there is an abstract operation based not on integers but on a finite representation for those values such as an interval. An interval is an abstraction that represents the values of real variables in a program. For example, the interval 1-7 represents integers 1, 3, 5, 7.

Brat and Venet launched their project by applying the state-of-the-art PolySpace static analysis tool to real NASA code from the space station, Deep Space 1, and Mars PathFinder. The tool worked just well enough to set baseline measurements for scalability and precision—that is, the proportion of definite classifications to warnings of possible problems in the code. Then they created a specialized tool tailored for NASA software as they knew it. They have begun testing their new tool—the C Global Surveyor—on entry, descent, and landing software for the Mars Exploration Rover as well as on software for the rover itself. Along

with other researchers, they have begun exploring ways in which MDS developers might make gains in terms of automated verification by accepting some restrictions on how they program in C++.

*Mars Science Laboratory—artist's conception. Courtesy NASA.*

## TO BE HUMAN IS TO ENGINEER

Kirk Reinholtz, one of the key collaborators working with Holzmann and other proponents of automated approaches, is an eloquent spokesman for a different point of view. Reinholtz, who describes himself as "a software engineer, born and raised," is the chief programmer on the Mars Science Laboratory project.

In explaining how his team is building one million-plus lines of Mission Data System code into a multipurpose system for the 2009 launch, Reinholtz shows a firm faith in the power of human skills, judgment, and processes: "The purpose of that software is essentially to turn everything we've learned over decades of doing embedded, extremely reliable software into more of an engineering discipline than an art."

According to Reinholtz, the genius of MDS lies in "hoisting" issues that experience shows can be real problems in actual missions to the architectural level, "where we have fairly mature processes to observe them, get plenty of eyes on it, do the verification, and so forth." What a contemporary software engineer might tend to hide, he says, MDS makes explicit.

Down to the level of specifying a vocabulary for discussing engineering goals, MDS aims to make it easier for programmers to do the right thing and harder for them to make mistakes.

The approaches that MDS and model checking typify—one more confident in experience and ingenuity, the other more wary of human fallibility—offer solutions that could contribute to the success of the Mars Science Laboratory and other missions. Both also may point toward a way out of what JPL principal software architect Nicolas Rouquette calls "the traditional divorce and death march," in which projects succeed because of the sacrifice of "heroes."

## DESIGNING NEXT-GENERATION SYSTEMS

Traditional design methods—including those used at JPL in the past—involve a hierarchical decomposition of a system into subsystems, with each software engineering team providing its own customized solutions and iteratively integrating them at the system level.

Although subsystem decomposition works in well-understood domains such as enterprise systems, it works less well in an environment such as deep space where system resources such as power or memory are limited and the system must interact with an unpredictable physical environment.[1,5] These various subsystems must share these limited resources, but the assumptions that one subsystem design team makes might not hold across all subsystems. For example, it is virtually impossible to abstract an idealized camera in this environment because the physical camera draws power used by other resources, consumes CPU cycles to process data, and could be used as a navigation device to track stars or celestial bodies. In the end, says Daniel Dvorak, MDS deputy architect, using object-oriented methodologies in such an environment leads to a hierarchical subsystem decomposition that is difficult to verify, validate, or reuse.

Drawing on his vast experience with past JPL missions, Robert Rasmussen defined in MDS an abstract architecture for designing next-generation deep-space systems. Such systems interact with the physical world, reacting to and trying to control physical state, and mission controllers on the ground think in similar terms. Therefore, Rasmussen proposed a state-based control architecture that models the interactions between physical states and attempts to control state by applying goals and constraints on state rather than employing a time-based sequence of commands, thus allowing varying degrees of mission autonomy.

Unlike traditional object-oriented subsystem decomposition, which hides or encapsulates system interactions in local program variables, flags, counters, pointers, and if statements, MDS elevates state variables to the top of the architectural hierarchy where they can be seen, understood, and engineered.[1,6] As such, systems engineering becomes state analysis, with a precise and bounded vocabulary for describing all interactions between the elements the project comprises. This vocabulary includes terms such as state variables, estimators, controllers, state value histories, state effects models, measurements, time and state constraints, resources, commands, scenario fragments, and goals.

Just as important, this constrained vocabulary translates to the software engineering domain, providing strictly defined input/output values that either constrain or aid a system developer when implementing that portion of the system: Either the programmer has introduced unnecessary com-

**MDS provides a domain-specific framework for a family of applications and their attendant flight, ground, and test platforms.**

plexity or a potential error, or the initial systems analysis was wrong. Moreover, all hardware components, their attributes, and the modeled transformations between them can be captured in a state database and queried by mission planners and systems engineers; by software developers and their automated tools; by simulators testing software against undelivered hardware; and, perhaps more important, by onboard software in the case of unexpected system behavior.

Covering the entire engineering discipline and enforcing it from top to bottom, MDS uses a shared vocabulary based on state analysis for requirements capture. It applies this vocabulary to a software development process engineered down to the level of configuration management, at the same time leaving room for new technical approaches.

MDS combines a state-based systems architecture and component-based software architecture to provide a domain-specific framework for a family of applications—whether they are multiple generations of satellites, landers, or rovers—and their attendant flight, ground, and test platforms.[1] The state-based architecture offers a structured process for disciplined analysis that emphasizes model-based design for estimation and control, makes interactions explicit, and exposes complexity. The component architecture provides frameworks and adapter's guides, reusable building blocks in object-oriented design, guides for how to adapt it for concrete tasks, and examples of framework usage.

Flight software is largely embedded, whereas ground software has extensive resources in the form of servers, adequate power, and so forth. But whether it is one millisecond or one day, a communication delay closes the control loop from both the architectural and customer viewpoints. For example, mission controllers view the spacecraft as a point in a two-dimensional plot, whereas the spacecraft itself is an object in three-dimensional space. Then the question becomes, where is it pointing?

Both viewpoints use the same mathematical equations but derive different results due to their different degrees of freedom. In contrast, because MDS focuses on the similarities rather than the differences, developers can use the same mathematical and architectural framework to write both the flight navigation and ground control software. Unlike past missions, JPL will cost together the flight and ground software for the Mars Science Laboratory—a major cultural change.

## STATE ANALYSIS

State analysis as embodied in MDS provides a uniform, methodical, and rigorous approach to discovering, characterizing, representing, and documenting a system's states, and modeling their behavior and the relationships among them. Knowledge of the system and its environment is represented over time in state variables, which include such things as

- *dynamics*—vehicle position and attitude, gimbal angles, wheel rotation;
- *environment*—ephemeris, light level, atmospheric profiles, terrain;
- *device status*—configuration, temperature, operating modes, failure modes;
- *parameters*—mass properties, scale factors, biases, alignments, noise levels;
- *resources*—power and energy, propellant, data storage, bandwidth;
- *data product collections*—science data, measurement sets;
- *data management and transport policies*—compression, deletion, transport priority; or
- *externally controlled factors*—spacelink schedule and configuration.

MDS reports, stores, and transports information about the system as histories of state, measurements, and commands.

Systems engineers use state analysis to capture mission objectives in detailed scenarios, keep track of system constraints and operating rules, describe the methods they will use to achieve objectives, and record information about hardware interfaces and operation. Throughout, the common framework elements (vocabulary) unify all aspects of the design process. For example, if the *goal* is to move a rover to a rock, the *state variable* to be controlled is the rover's position relative to the rock. *Measurements* provide evidence for that state—for example, wheel rotations, sun sensor, or stereo camera. For a stereo camera, *measurement models* indicate the distance to terrain features, light level, camera power (on/off), camera health, and so on.

Figure 1 shows the MDS goal-oriented state-based architecture. Given a model of how things work, estimators find "good" explanations for measurement (sensor) and command (actuator) data to estimate state. State variables hold state values, including the degree of uncertainty. To describe state evolution, state timelines combine current and past estimates with future predictions and plans. Together, time-based state information and mod-

els of state behavior supply the information needed to operate a system and assess performance.[1]

Operators express their intent in the form of goals declaring what should happen—not how. The operators and planners can elaborate the goals recursively, and even conditionally, into lower-level goals that are coordinated by a controller/scheduler that uses priority as the final arbiter to resolve conflicts. MDS keeps state estimations and state control completely separate to avoid the temptation to warp a state estimation to meet a control objective or the risk of having multiple interpretations for the same data.[1]

Finally, MDS represents actual hardware components in software as hardware adapters that facilitate the delineation of the abstract system model (including time) by translating raw input/output data and measurement and command models into abstract declarations about state. In addition, software-based hardware adapters can augment system hardware with supplemental behaviors such as sampling, I/O, sequencing and synchronization, time and metadata tagging, data buffering and routing, data format translation, error checking, and data preprocessing and compression. More important still, MDS isolates state frameworks from platform-specific interfaces and supports real, simulated, or abstract hardware in real or virtual time.

## MISSION POSSIBLE?

Elevating state to a first-class entity and controlling state through goals that constrain it over time will guarantee an ambitious agenda for MDS. Because MDS makes both state and the models that describe it explicit, goals are self-checking by nature, prescribing possible inputs and outputs. Goals also provide hooks for model checkers, code instrumentation, and verification and validation tools to increase reliability.

A more intriguing possibility proposed by JPL's Rouquette would use automatic code generation for the vast majority of spacecraft system software. Rouquette views current model checking techniques as more suitable for niche problems, algorithms, and carefully chosen pieces of code where developers know what they're looking for—not for abstracted sections of code from large-scale systems in which complexity arises from the interactions between the different parts. Instead, Rouquette is banking on code generation and transformation techniques already used successfully in Deep Space 1.

Rouquette used the MathWorks' Stateflow toolbox and some homegrown tools to generate 90 percent of the code for the DS1 fault-protection subsystem from 60 or so state machines. Since building such models can take several months and they can be revised 50 or more times, automatically generating efficient C code with a small code footprint was a huge win—allowing talented engineers to build the system via models, as in civil engineering, rather than waiting for programmers to code, verify, and, more often than not, recode the particular system repeatedly. However, this solves only half the problem.

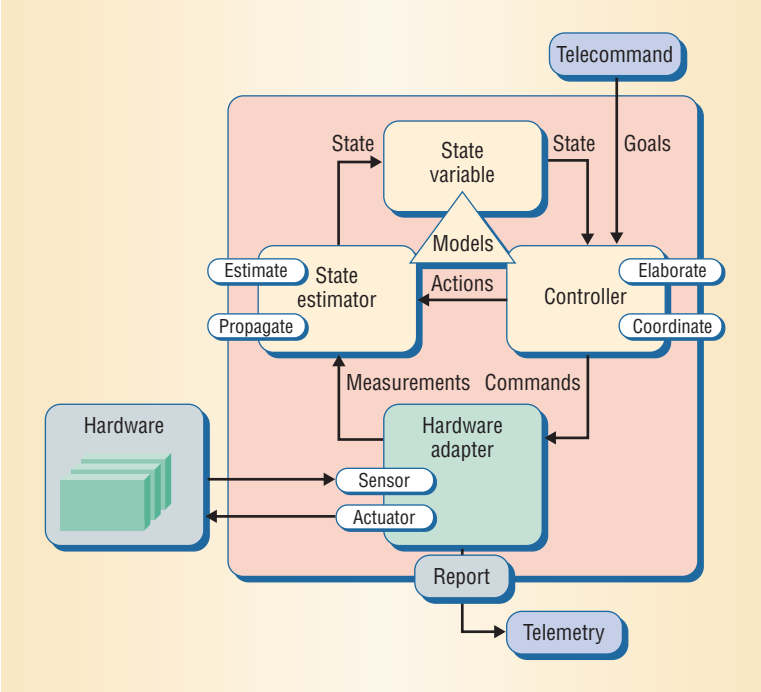



***Figure 1. MDS architecture. A goal is a constraint on the value of a state variable over a time interval. State variables hold state values, including degree of uncertainty. Models express mission-specific relations among states, commands, and measurements. Estimators interpret measurement and command evidence to estimate state. Controllers issue commands, striving to achieve goals. Hardware proxies provide access to hardware buses, devices, and instruments.***

Over the past decade, code generators have indeed become more clever, progressing from "dumb" programming tools that do simple string manipulation to something more akin to a compiler that can parse a specification to gain a notion of valid input and then perform internal processing to produce some output. While these more sophisticated tools now do a very good job on the input side, says Rouquette, there is far less assurance on the output side, especially when producing output in languages whose meager semantics are difficult to prove formally and hard for model checkers to check.

With its state-based models, MDS offers a more formal expression of valid input and output. Smarter languages with higher levels of abstraction will facilitate model checking and make it possible to apply modern compiler technology and algorithms such as pattern matching, tree recognition, and algebraic formalisms for operations on trees to code generators themselves.

Rouquette's grand vision is to generate 95 percent of the Mars Science Laboratory code automatically, leaving only a small runtime that will be coded manually and will require extensive model checking and testing for validation. Space is a

largely unknown environment, says Rouquette, and the only way to ensure mission success is by having the many bright people involved use their knowledge, imagination, and time to engineer systems instead of writing code. "Analyzing rocks on Mars sets our expectations way too low, and there are many other nice places to go. But we have to make these things dirt cheap and a sure thing."

By exploring new technologies and approaches to develop provably reliable software within tough constraints, NASA has a chance to deliver even more than the results of the scientific probes it launches. "We're the engineers," Tom Gavin says, "but we do this in the name of science." Whatever progress the space agency makes in addressing its own software issues seems likely to advance the state of the art, contributing to computer science as well as software engineering. In addition, any successful spin-off that improves reliability while cutting development time and costs could, in principle, generate savings for US industry equal to the nation's budget for space exploration. ■




## Acknowledgments

We thank Erik Antonsson, Matthew Barry, Guillaume Brat, Daniel Dvorak, Martin Feather, Erann Gat, Tom Gavin, Dimitra Giannakopoulou, Klaus Havelund, Gerard Holzmann, Michael Lowry, Kenny Meyer, David Nichols, Tom Prince, Kirk Reinholtz, Nicolas Rouquette, and Willem Visser for their time. In addition, we owe a great debt to Lisa Townsend at JPL and Michael Mewhinney and John Bluck at Ames for arranging the interviews with these researchers.


## References


1. D. Dvorak et al., "Software Architecture Themes in JPL's Mission Data System," *Proc. AIAA Space Technology Conf. and Exposition*, 1999; http://x2000.jpl.nasa.gov/nonflash/publications/aiaa99_mds_final.pdf.
2. G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
3. K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder," *Int'l J. Software Tools for Technology Transfer*, vol. 2, no. 4, 2000, pp. 366-381.
4. W. Visser et al., "Model Checking Programs," *J. Automated Software Engineering*, vol. 10, 2002, pp. 203-232; http://ase.arc.nasa.gov/jpf/papers.html.
5. K. Havelund and G. Rosu, "Monitoring Java Programs with Java PathExplorer," *Proc. Workshop Runtime Verification, Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, 2001.
6. D. Dvorak, "Challenging Encapsulation in the Design of High-Risk Control Systems," presentation, *Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, 2002.



***Patrick Regan*** *is senior correspondent for science and technology at* NJN News *on New Jersey's public television and radio network. Contact him at patrick.regan@ieee.org.*

***Scott Hamilton*** *is* Computer*'s senior acquisitions editor. Contact him at s.hamilton@computer.org.*