Within two hours following launch, though, most parts of the launch vehicle had been discarded. At that point, the remaining main parts of the spacecraft included the cruise-stage, the backshell with a large parachute inside, the descent-stage with its intricate sky crane mechanism, the rover, and a large heat shield (see Figure 1).

The cruise-stage was equipped with solar panels to help power the spacecraft during its nine-month trip to Mars, as well as a star tracker to help with navigation, and thrusters to perform small course corrections. All were cast off approximately 10 minutes before the spacecraft entered the Martian atmosphere.

The remaining parts were now all contained within the backshell and protected by the heat shield. The backshell, large enough to hold a small car, had its own set of thrusters to make small course adjustments during the hypersonic entry into the Martian atmosphere. During entry, the backshell cast off several large chunks of ballast mass (weighing some 320 kg) to adjust the center of gravity for the landing at the command of the rover computer that controls the entire mission.

Approximately three minutes before landing the parachute deployed to slow the spacecraft from 1,500 km/h to 300 km/h. The heat shield was ejected, and less than a minute before touchdown the descent stage dropped away from the backshell (see Figure 2). From this point it was up to the descent stage to guide the rover, with wheels deployed, to the surface (see Figure 3), disconnect itself, and fly away a safe distance to crash. All steps in this sequence were again con-
Elvis has Spirit. The answer is 42...END\r\n" /* 1095 with NULL */
The software that controls an interplanetary spacecraft must be designed to a high standard of reliability; any small mistake can lead to the loss of the mission and its unique opportunity to expand human knowledge.

Extraordinary measures are taken in both hardware and software design to ensure spacecraft reliability and that the system can be debugged and repaired from millions of miles away.

Formal methods help verify intricate software subsystems for the potential of race conditions and deadlocks; new model-checking techniques automate the verification process.

Reducing Risk

There are standard precautions that can help reduce risk in complex software systems. This includes the definition of a good software architecture based on a clean separation of concerns, data hiding, modularity, well-defined interfaces, and strong fault-protection mechanisms. It also includes a good development process, with clearly stated requirements, requirements tracking, daily integration builds, rigorous unit and integration testing, and extensive simulation.

This article does not revisit these well-known principles of software design. Instead, it focuses on a different set of precautions the flight software team took in the development of the MSL mission software that is perhaps less common. We restrict ourselves here to three specific topics: First, the coding standard we adopted, which is distinguished by being sparse, risk-based, and supported by automated compliance-checking tools; second, the redefined code-review process we adopted, which allows us to thoroughly scrub large amounts of code efficiently, again leveraging the use of tools; and third, logic model-checking tools to formally verify mission-critical code segments for the existence of concurrency-related defects.

Risk-based coding rules. No method can claim to prevent all mistakes, but that does not mean we should not try to reduce their likelihood. Before we can do so, though, we have to know what types of mistakes occur controlled by one of two available computers located within the body of the rover itself.

With each new mission flown to Mars, the size and complexity of both spacecraft hardware and software has increased. The Mars Science Laboratory (MSL) mission, for instance, uses more code than all previous missions to Mars combined, from all countries that have tried to do it. This rapid growth in the size of the software is clearly a concern, but one not unique to this application domain. Unlike most other software applications, though, the embedded software for a spacecraft is designed for a one-of-a-kind device with an uncommon array of custom-built peripherals. The code targets just one user (the mission), and for the most critical parts of the mission the software is used just once, as in the all-important landing phase, which lasts only minutes. Moreover, the software can be frustratingly difficult to test in an accurate representation of the environment in which it must ultimately operate, yet there are no second chances. The penalty for even a small coding error can be not just the loss of a rare opportunity to expand our knowledge of the solar system, it can also mean the loss of a significant investment and put a serious dent in the reputation of the responsible organization.

Reducing Risk

There are standard precautions that can help reduce risk in complex software systems. This includes the definition of a good software architecture based on a clean separation of concerns, data hiding, modularity, well-defined interfaces, and strong fault-protection mechanisms. It also includes a good development process, with clearly stated requirements, requirements tracking, daily integration builds, rigorous unit and integration testing, and extensive simulation.

This article does not revisit these well-known principles of software design. Instead, it focuses on a different set of precautions the flight software team took in the development of the MSL mission software that is perhaps less common. We restrict ourselves here to three specific topics: First, the coding standard we adopted, which is distinguished by being sparse, risk-based, and supported by automated compliance-checking tools; second, the redefined code-review process we adopted, which allows us to thoroughly scrub large amounts of code efficiently, again leveraging the use of tools; and third, logic model-checking tools to formally verify mission-critical code segments for the existence of concurrency-related defects.

Risk-based coding rules. No method can claim to prevent all mistakes, but that does not mean we should not try to reduce their likelihood. Before we can do so, though, we have to know what types of mistakes occur controlled by one of two available computers located within the body of the rover itself.

With each new mission flown to Mars, the size and complexity of both spacecraft hardware and software has increased. The Mars Science Laboratory (MSL) mission, for instance, uses more code than all previous missions to Mars combined, from all countries that have tried to do it. This rapid growth in the size of the software is clearly a concern, but one not unique to this application domain. Unlike most other software applications, though, the embedded software for a spacecraft is designed for a one-of-a-kind device with an uncommon array of custom-built peripherals. The code targets just one user (the mission), and for the most critical parts of the mission the software is used just once, as in the all-important landing phase, which lasts only minutes. Moreover, the software can be frustratingly difficult to test in an accurate representation of the environment in which it must ultimately operate, yet there are no second chances. The penalty for even a small coding error can be not just the loss of a rare opportunity to expand our knowledge of the solar system, it can also mean the loss of a significant investment and put a serious dent in the reputation of the responsible organization.

Reducing Risk

There are standard precautions that can help reduce risk in complex software systems. This includes the definition of a good software architecture based on a clean separation of concerns, data hiding, modularity, well-defined interfaces, and strong fault-protection mechanisms. It also includes a good development process, with clearly stated requirements, requirements tracking, daily integration builds, rigorous unit and integration testing, and extensive simulation.

This article does not revisit these well-known principles of software design. Instead, it focuses on a different set of precautions the flight software team took in the development of the MSL mission software that is perhaps less common. We restrict ourselves here to three specific topics: First, the coding standard we adopted, which is distinguished by being sparse, risk-based, and supported by automated compliance-checking tools; second, the redefined code-review process we adopted, which allows us to thoroughly scrub large amounts of code efficiently, again leveraging the use of tools; and third, logic model-checking tools to formally verify mission-critical code segments for the existence of concurrency-related defects.

Risk-based coding rules. No method can claim to prevent all mistakes, but that does not mean we should not try to reduce their likelihood. Before we can do so, though, we have to know what types of mistakes occur
most often in this domain. Finding the data is not difficult. Most anomalies that have affected space missions are carefully studied and documented, with most information publicly available. We used it to categorize the root causes of each software anomaly to produce a list of the primary areas of concern.

Among them are basic coding and design errors, especially those caused by an undisciplined use of multitasking. Other frequently occurring errors originate in the use of dynamic memory-allocation techniques, which in the early days of space exploration often meant the use of dynamic memory overlays. Finally, the data also shows even standard fault-protection techniques can have unintended side effects that can also cause missions to fail.

The coding standard we developed based on this study differs from many others in that it contained only risk-related, as opposed to style-related, rules. Our view is that coding style (for instance, where curly brackets are placed and how a loop statement is formatted) can be adjusted easily to the preferences of a viewer (or reviewer) using standard code-reformatting tools. Risk-reduction, though, is a consideration that should trump formatting decisions. We used two criteria for inclusion of rules in our new JPL coding standard: First, the rule had to correlate directly with observed risk based on our taxonomy of software anomalies from earlier missions; and second, compliance with the coding rule had to be verifiable with tool-based checks.

Compliance with a coding standard need not be an all-or-nothing proposition; not all code is equally critical to an application. The coding standard we developed therefore recognizes different levels of compliance that apply to different types of software (see Figure 4).

Level-one compliance, or LOC-1, sets a minimal standard of workmanship for all code written at JPL. There are just two rules at this level: The first says all code must be language compliant; that is, it cannot rely on compiler-specific extensions that go outside the language definition proper. For flight software the language standard used at JPL is ISO-C99. The second rule at this level requires that all code can pass both the compiler and a good static source code analyzer without triggering warnings. For this test, the compiler is used with all warnings enabled.

LOC-2 compliance adds rules that are meant to secure predictable execution in an embedded system context. One important rule defined at this level is that all loops must have a statically verifiable upper bound on the number of iterations they can perform.
To reach LOC-3 compliance, one of the most important rules concerns the use of assertions. We originally formulated the rule to require all functions with more than 10 lines of code contain at least one assertion. We later revised it to require that the flight software as a whole, and each module within it, had to reach a minimal assertion density of 2%. There is compelling evidence that higher assertion densities correlate with lower residual defect densities.\textsuperscript{14} The MSL flight software reached an overall assertion density of 2.26%, a significant improvement over earlier missions. This rate also compares favorably with others reported in the literature.\textsuperscript{1,7} One final departure from earlier practice was that on the MSL mission all assertions remained enabled in flight, whereas before they were disabled after testing. A failing assertion is now tied in with the fault-protection system and by default places the spacecraft into a predefined safe state where the cause of the failure can be diagnosed carefully before normal operation is resumed.

LOC-4 is the target level for all mission-critical code, which for the MSL mission includes all on-board flight software. Compliance with this level of the standard restricts use of the C preprocessor, as well as function pointers and pointer indirections. The cumulative number of coding rules that must be complied with to reach this level remains relatively low, with no more than 31 risk-related rules.

Safety-critical and human-rated software is expected to comply with the higher levels of rigor defined in LOC-5 and LOC-6. These two highest levels of compliance add all rules from the well-known MISRA C coding guidelines\textsuperscript{16} not already covered at the lower levels.

We worked with vendors of static source code analysis tools, including Coverity, Codesonar, and Semmle, to develop automatic compliance checkers for the majority of the rules in our coding standard. Compliance with all risk-based rules could therefore be verified automatically with multiple independent tools on every build of the MSL software.

One additional precaution we undertook starting with the MSL mission was to introduce a new certification program for flight-software developers, allowing us to, for instance, discuss the detailed rationale for all coding rules and reinforce knowledge of defensive coding techniques. The certification program is concluded with an exam, passage of which is required for all developers who write or maintain spacecraft software.

**Tool-based code review.** Not all software defects can be prevented by even the strongest coding rules, meaning it is important to devise as many methods as possible to intercept the defects that slip through and use them as early and often as possible. One standard mechanism for scrutinizing software is peer code review. Traditionally, in a peer-code-review session, expert developers are invited to provide feedback in a guided code walkthrough. This process can work exceptionally well, but only for relatively small amounts of code. If more than a few hundred lines of code are examined in a single session, the effectiveness of the session, measured by number of flaws exposed, decreases rapidly. Reviewing a few million lines of code in this manner would severely strain the system, if not the reviewers.\textsuperscript{8}

Peer reviewers can excel at identifying design flaws but are much less reliable at the more down-to-earth job of checking for mundane issues like rule-compliance and avoidance of common coding errors. Fortunately, this is where static source-code-analysis tools can prove their value. A static analyzer will not tire of checking for the same types of defects over and over, night after night, patiently reporting all violations. We have therefore made extensive use of this technology.

A wide range of commercial static source-code-analysis tools is on the market, each with slightly different strengths. We found that running multiple analyzers over the same code can be very effective; there is surprisingly little overlap in the output from the various tools. This observation prompted us to run not just one but four different analyzers over all code as part of the nightly integration builds for the MSL mission.

The analyzers we selected—Coverity, Codesonar, Semmle, and Uno—had to be able to identify likely bugs with a reasonably low false-positive rate, handle millions of lines of code efficiently, and allow for the definition of custom checks (such as verifying compliance with the rules from our coding standard). The output of each tool was uniformly reformatted with simple post-processing scripts so all tool reports could be made available within a single vendor-neutral code-review tool we developed, called Scrub. The Scrub tool was designed to integrate the output of the static analyzers and any other type of background checkers with human-generated peer code review comments in a single user-interface.\textsuperscript{8}

In peer code reviews, the reviewers are asked to add their observations to the code in the Scrub tool, which is prepopulated with static analysis results from the most recent integration build of the code. The module owner is required to respond to each report, whether generated by a human peer reviewer or by one of the static analysis tools. To respond, the Scrub tool allows the module owner to choose from three possible responses: agree, meaning the module owner accepts the comment and agrees to change the code to address the concern; disagree, meaning the module owner has reason to believe the code as written should not be changed; and discuss, meaning the comment or report is unclear and needs clarification before it can be addressed (see Figure 5).

The peer code reviews, and the re-
Peer reviewers can excel at identifying design flaws but are much less reliable at the more down-to-earth job of checking for mundane issues like rule-compliance and avoidance of common coding errors.
summary of a proof of correctness. A few years following its publication an attempt was made to formalize that proof with a theorem prover as part of a master’s thesis project. The formalization revealed that both the original proof and the algorithm were flawed. A correction to the algorithm could be proven correct with the theorem prover. Each proof attempt, for both the original algorithm and the corrected version, reportedly took several months.

Lamport later formalized the original algorithm in +CAL, showing the flaws could be found more quickly through a model checker. Lamport noted the proof with the TLA+ model checker could be completed in less than two days, most of which was needed to define a formal model of the original algorithm in the language supported by the model checker.

As shown here, a model extractor can help avoid the need for manual construction of a formal model as well, allowing us to perform these types of verification on multithreaded code fragments in minutes instead of days. We use the original algorithm from Detlefs to show how this verification approach works. With it, finding the flaw in the implementation of the algorithm requires no more than typing in a few lines of text and executing a single command.

The algorithm uses an atomic Double-word Compare-And-Swap, or DCAS, instruction; Figure 6 gives the semantics of this instruction as defined in Detlefs. Figure 7 reproduces two C routines from Detlefs for adding an element to the right of the queue and for deleting an element from the same side. The routines for adding or deleting elements from the left side of the queue are symmetric. The node structure used has three fields: a left pointer L, a right pointer R, and an integer value V.

To verify the code we first define a simple test driver that exercises the code by adding and deleting elements (see Figure 8). For simplicity, this example uses only the pushRight() and popRight() routines.

In the example test driver in Figure 8, the writer initializes the queue on line 74, and the reader waits until this step is completed on lines 57–59.
reader contains an assertion on line 64 to verify the values sent by the writer are received in the correct order, without omissions.

We can perform the test using different threads for the reader and the writer, though these tests alone cannot establish the correctness of the algorithm. A model checker is designed to perform this type of check more rigorously. If there is any possible interleaving of the thread executions that can trigger an assertion failure, the model checker is guaranteed to find it. To use the model checker, we define a small configuration file that identifies the parts of the code we are interested in. This configuration file allows us to define an execution context for the system we want to verify by extracting the relevant parts of the code and placing them into an executable system that is then analyzed.

Figure 9 shows the complete configuration file needed to verify this application. The first four lines identify four functions in source file dcas.c we are interested in extracting as instrumented function calls. The next two lines identify sample_reader and sample_writer as active threads that will call these functions. The last three lines in the configuration file define the required header file dcas.h that holds the definition of data structure Node and the name of the source file (dcas.c) to which the verifier must be linked for additional routines, including a C encoding of the function that defines the semantics of the DCAS instruction (also shown in Figure 6).

The verification of the algorithm can now be performed with a single command, using the model-extraction tool Modex and the model checker Spin (see Figure 10).

The command takes approximately 12 seconds of real time to execute, of which only 0.02 seconds is needed for the verification itself. The rest of the runtime is taken by the model extractor to generate the verification model from the source code, for Spin to convert that model into optimized C code, and finally for the C compiler to produce the executable that performs the verification. None of these steps requires further user interaction.

A replay of the error-trail reveals a race condition that can lead to an assertion violation and therefore shows the algorithm to be faulty (see figures 11, 12, and 13). Statements executed by the writer process are marked with \texttt{W} and statements executed by the reader process with \texttt{R}. First consider Figure 11. After the initial call to initialize in the sample_writer routine (line 74 in Figure 8), the writer initiates its first call to pushRight on line 77, with value 0. This value is then stored by executing lines 7 through 19 in the pushRight routine.

The next statement in the execution of pushRight would now be a call on DCAS to complete the update, but that call is delayed. Meanwhile, the sample_reader is free to proceed with calls to popRight to poll the queue for new elements (see Figure 12). The first call (line 62 in Figure 8) succeeds for new elements (see Figure 12). The remaining steps in Figure 12 illustrate the execution of the popRight routine for that call.

Figure 9. Modex configuration file.

```
%X -e pushRight
%X -e popRight
%X -e initialize
%X -e dcas_malloc
%X -a sample_reader
%X -a sample_writer
%D
#include "dcas.h"
%0 dcas.c
```

Figure 8. C code for a sample test driver.

```c
53 void
54 sample_reader(void)
55 { int i, rv;
56 60 while (!RH)
57 { /* wait */
58   for (i = 0; i < 10; i++)
59   {    /* wait */
60     while (!RH)
61     {   i--;
62   }
63         {   i--;
64         } else
65         {   assert(rv == i);
66         if (rv != EMPTY)
67         {    /* wait */
68             {    /* wait */
69         }
70 74 void
71 sample_writer(void)
72 { int i, v;
73 74 initialize();
75 76 for (i = 0; i < 10; i++)
77 {   v = pushRight(i);
78   if (v != OKAY)
79   { i--;}
80 81 }
```
This call should not succeed because the `pushRight` call, initiated by the writer in Figure 11, has not yet completed its update. But the trap has now been set. The `sample_reader` thread now moves on to the next call, after incrementing the value of `i`. This second call to `popRight` completes the same way it did before and again returns the value `0`, resulting in the failure (see Figure 13).

The model-extraction method used here is defined in such a way it allows for very simple types of instrumentation in basic applications. The model extractor always preserves the application’s original control flow. However, it also supports the definition of more advanced abstraction functions in configuration files (similar to the one in Figure 9) that can be used to reduce the complexity of extracted models. The default conversion rule, which defines a one-to-one mapping of statements from the source code into the model, allows for direct verification of a surprisingly large set of multithreaded C programs and algorithms.

The MSL mission made extensive use of this automated capability to verify critical multithreaded algorithms, directly using their implementation in C. For larger subsystems, we also manually constructed Spin verification models in a more traditional way and analyzed them. The largest such MSL subsystem was a critical data-management module implemented in approximately 45,000 lines of C. The design of this subsystem was converted manually into a Spin verification model of approximately 1,600 lines, in close collaboration with the module designer.

In most cases, the model-checking runs successfully identified the existence of subtle concurrency flaws that could be remedied in the software. For the file system software in particular, the model-checking runs became a routine part of our regression “tests,” executed after every change in the code, often surprising us with the ease with which it could identify newly committed coding errors.

**Conclusion**

The MSL spacecraft performed flawlessly in delivering Curiosity to the surface of Mars in August 2012 where it is currently exploring the planet (see Figure 14). The rover has meanwhile achieved its primary mission, which was to determine if our neighbor planet could in principle have supported life in the distant past.

Every precaution was taken to optimize the chances of success, and not just in the development of the software. Critical hardware components were duplicated, including the rover’s main CPU. But though it is not difficult to see how duplication of an essential hardware component helps improve system reliability, seeing how one can use redundancy to improve software reliability is less simple.

We gave two examples of how software redundancy was nonetheless used on the MSL mission. The first—emphasis on use of assertions throughout the code—may sound obvious but is rarely recognized as a pro-
Every precaution was taken to optimize the chances of success, and not just in the development of the software. Critical hardware components were duplicated, including the rover’s main CPU.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, under a contract with the National Aeronautics and Space Administration. Credit for the nearly flawless performance of the MSL flight software to date goes to the superb software development team that created, reviewed, analyzed, tested, and retested the code, working countless hours.

References


Gerard J. Holzmann (gholzmann@acm.org) is a senior research scientist and a fellow at NASA’s Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA.

© 2014 ACM 0001-0782/14/02 515.DD