DOI:10.1145/2560217.2560218

Redundant software (and hardware) ensured Curiosity reached its destination and functioned as its designers intended.

BY GERARD J. HOLZMANN

Mars Code

ON AUGUST 5, 2012,10:18 P.M. PST, a large rover named Curiosity made a soft landing on the surface of Mars. Given the one-way light-time to Mars, the controllers on Earth learned about the successful touchdown 14 minutes later, at 10:32 P.M. PST. As can be expected, all functions on the rover, and on the spacecraft that brought it to its destination 350 million miles from Earth, are controlled by software. This article discusses some of the precautions the JPL flight software team took to improve its reliability.

To begin the journey to Mars you need a launch vehicle with enough thrust to escape Earth's gravity. On Earth, Curiosity weighed 900 kg. It weighs no more than 337.5 kg on Mars because Mars is smaller than Earth. Curiosity began its trip atop a large Atlas V 541 rocket, which, together with fuel and all other parts needed for the trip, brought the total launch weight to a whopping 531,000 kg, or 590 times the weight of the rover alone.

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 on 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-

This image depicts the "fill-packet" transmitted by the Curiosity rover many times each sol (a day on Mars) whenever there is no useful telemetry to send to Earth. The fill packet lists 50 members of the NASA JPL flight software team as well as an in memoriam list of another 18, including the crew of the Challenger and Columbia shuttles and the astronauts killed in a pre-launch test for Apollo 1, and inspirational remarks from astronomer Carl Sagan.

```
/*
 * Fill packet.
 */
STATIC U8 dwn_fill_packet_data[DWN_MAX_LF_FILL_PKT_BODY] = /* 1095 total */
  " M.Allen E.Benowitz J.Biesiadecki Y.Brenman       \r\n"  /*   50 */
  " T.Canham J.Carsten B.Cichy K.Clark M.Clark       \r\n"  /*  100 */
  " K.Fleming D.Gaines G.Gandhi C.Garcia K.Gostelow  \r\n"  /*  150 */
  " H.Hartounian D.Helmick Q.Ho G.Holzmann R.Joshi   \r\n"  /*  200 */
  " W.Kim D.Lam D.Leang C.Leger J.Levison T.Litwin   \r\n"  /*  250 */
  " M.Maimone L.Manglapus T.Neilson C.Oda M.Pack     \r\n"  /*  300 */
  " P.Pandian G.Reeves S.Scandore M.Schoppers        \r\n"  /*  350 */
  " B.Shenker B.Smith D.Smyth J.Snyder M.Tuszynski   \r\n"  /*  400 */
  " T.Uchenik V.Verma K.Weiss C.Williams M.Yang      \r\n"  /*  450 */
  " P.Brugarolas L.Phan M.SanMartin F.Serricchio     \r\n"  /*  500 */
  " G.Singh A.Amed .. In Memoriam: Grissom Chaffee   \r\n"  /*  550 */
  " White Scobee Smith Resnik Onizuka McNair Jarvis  \r\n"  /*  600 */
  " McAuliffe Husband McCool Anderson Ramon Chawla   \r\n"  /*  650 */
  " Brown Clark Grammier                             \r\n"  /*  700 */
  "                                                  \r\n"  /*  750 */
  "   Exploration is in our nature. We began as      \r\n"  /*  800 */
  "   wanderers, and we are wanderers still. We      \r\n"  /*  850 */
  "   have lingered long enough on the shores of     \r\n"  /*  900 */
  "   the cosmic ocean. We are ready at last to      \r\n"  /*  950 */
  "   set sail for the stars.          --Carl Sagan  \r\n"  /* 1000 */
  "                                                  \r\n"  /* 1050 */
" Elvis has Spirit. The answer is 42....END\r\n";          /* 1095 with NULL */
```


/*
 * Fill packet.
 */
STATIC U8 dwn_fill_packet_data[DWN_MAX_LF_FILL_PKT_BODY] = /* 1095 total */
" M.Allen E.Benowitz J.Biesiadecki Y.Brenman \r\n" /* 50 */
" T.Canham J.Carsten B.Cichy K.Clark M.Clark \r\n" /* 100 */
" K.Fleming D.Gaines G.Gandhi C.Garcia K.Gostelow \r\n" /* 150 */
" H.Hartounian D.Helmick Q.Ho G.Holzmann R.Joshi \r\n" /* 200 */
" W.Kim D.Lam D.Leang C.Leger J.Levison T.Litwin \r\n" /* 250 */
" M.Maimone L.Manglapus T.Neilson C.Oda M.Pack \r\n" /* 300 */
" P.Pandian G.Reeves S.Scandore M.Schoppers \r\n" /* 350 */
" B.Shenker B.Smith D.Smyth J.Snyder M.Tuszynski \r\n" /* 400 */
" T.Uchenik V.Verma K.Weiss C.Williams M.Yang \r\n" /* 450 */
" P.Brugarolas L.Phan M.SanMartin F.Serricchio \r\n" /* 500 */
" G.Singh A.Amed .. In Memoriam: Grissom Chaffee \r\n" /* 550 */
" White Scobee Smith Resnik Onizuka McNair Jarvis \r\n" /* 600 */
" McAuliffe Husband McCool Anderson Ramon Chawla \r\n" /* 650 */
" Brown Clark Grammier \r\n" /* 700 */
" \r\n" /* 750 */
" Exploration is in our nature. We began as \r\n" /* 800 */
" wanderers, and we are wanderers still. We \r\n" /* 850 */
" have lingered long enough on the shores of \r\n" /* 900 */
" the cosmic ocean. We are ready at last to \r\n" /* 950 */
" set sail for the stars. --Carl Sagan \r\n" /* 1000 */
" \r\n" /* 1050 */
" Elvis has Spirit. The answer is 42....END\r\n"; /* 1095 with NULL */

» key insights

- **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.**

trolled 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.

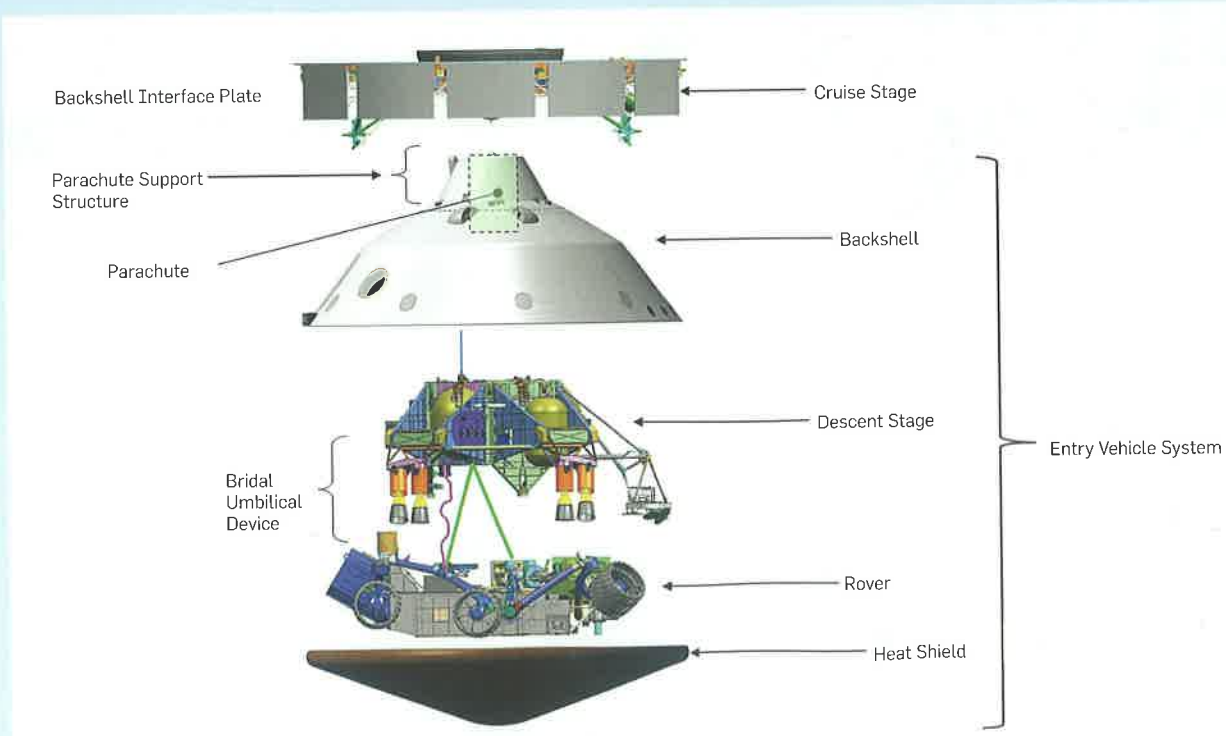
Reducing Risk

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.[18] 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 allowed 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

Figure 1. Spacecraft parts.



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.[9,13] 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).

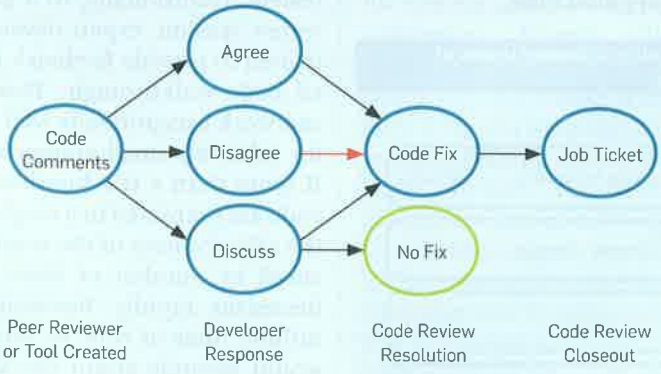
Level-one compliance, or LOC-1, sets a minimal standard of workmanship for all code written at JPL. There are just two rules at this level: The first says all code must be language compliant; that is, it cannot rely on compiler-specific extensions that go outside the language definition proper. For flight 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.

Figure 2. MSL descent stage.

Figure 3. MSL sky crane.

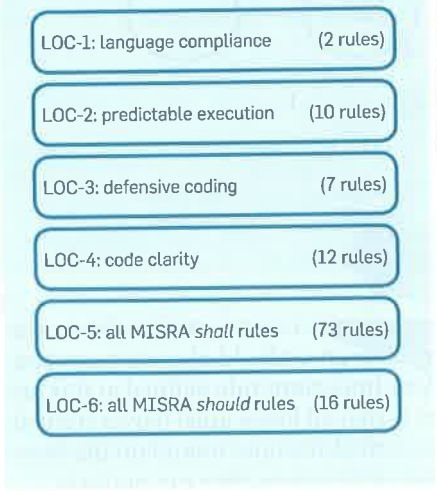
Figure 4. Life cycle of a code comment; orange arrow indicates where the developer disagrees with a code change but is overruled in the final review.



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.[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.[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.

Figure 5. Coding standard levels of compliance.



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[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.[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.[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-

sponses to all comments and reports, are done offline, outside meetings. Just one face-to-face meeting per module code review is used to resolve disagreements, clarify reports, and reach consensus on the changes to the code that have to be made.

In 145 code reviews held between 2008 and 2012 for MSL flight software, approximately 10,000 peer comments and 30,000 tool-generated reports were discussed.[20] Approximately 84% of all comments and tool reports led to changes in the code to address the underlying concerns. There was less than 2% difference in this rate between the peer-generated and the tool-generated reports. Explicit *disagree* responses from the module owner occurred in just 12.3% of the cases. The responses were overruled in the final code review session in 33% of those cases, leading to a required fix anyway. A *discuss* response was given for just 6.4% of all comments and reports, leading to a change in the code in approximately 60% of those cases.

These statistics from the MSL code-review process illustrate that the large majority of comments and tool reports led to immediately agreed-upon changes to the code and did not require discussion in the code review close-out meetings. The time saved allowed us to push the code-review process further than would have been possible otherwise. Critical modules, for instance, could now be reviewed multiple times before the code was finalized for launch.

Model checking. The strongest type of check we have in our arsenal for analyzing multithreaded code is logic model checking. The code for the MSL mission makes significant use of multithreading, with 120 parallel tasks being executed under the control of a real-time operating system. The potential for race conditions therefore always exists and has been a significant cause of anomalies on earlier missions. To thoroughly analyze the code for race conditions, we made extensive use of the capabilities of the logic model checker *Spin*,[10] together with an extended version of a model extraction tool for C code.[12]

Spin was developed in the Computing Science Research group of Bell Labs starting in the early 1980s and has been freely available since 1989. We earlier used this tool on the verification of key parts of the control software for a number of spacecraft, including Cassini,[21] Deep Space One,[5,6] and the Mars Exploration rovers.[11] We also used it in the recent investigation of possible triggers for unintended acceleration in Toyota vehicles.[17] In almost all these cases, the verification effort succeeded in identifying unsuspected software defects, especially concurrency-related issues that would be very difficult to uncover by other means.

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.

The model checker *Spin* specifically targets verification of distributed-systems software with asynchronous threads of execution. Its internal verification algorithm is based on Vardi and Wolper's automata-theoretic verification method.[23] Informally, *Spin* takes the role of a demonic process scheduler, trying to find system executions that violate user-defined requirements. Simple examples of the type of requirements that can be proven or disproven this way are the validity of program assertions and the absence of deadlock scenarios. But the model checker can also reach farther by verifying more complex requirements on feasible or infeasible program executions that can be expressed in linear temporal logic.[19]

We analyzed several critical software components for the MSL mission, including a dual-CPU boot-control algorithm (the algorithm that controls which of two available CPUs will take control of the spacecraft when it boots), the nonvolatile flash file system, and the data-management subsystem. Several vulnerabilities identified through these analyses could be eliminated from the code before the mission was launched, effectively helping reduce the risk of in-flight surprises. The basic procedure of software model checking, using the tools we developed, can be illustrated with a small example. (Because NASA rules prevent us from publishing actual flight code from the rover, we use equivalent public-domain code for this example.)

It can be unreasonably difficult to prove manually that a concurrent algorithm is correct under all possible execution scenarios. We take as our example a non-blocking algorithm for two-sided queues presented in Detlefs et al.[2] together with a four-page

Figure 6. Semantics of the DCAS instruction.

```
boolean DCAS (val *addr1, val *addr2,
              val  old1,  val  old2,
              val  new1,  val  new2)
{
    atomically {
        if (*addr1 == old1 && *addr2 == old2)
        {   *addr1 = new1;
            *addr2 = new2;
            return true;
        } else
        {   return false;
    }   }
}
```

Figure 7. C code for pushRight and popRight routines.

```
 1 Node *Dummy, *LH, *RH;
 2
 3 val
 4 pushRight(val v)
 5 {   Node *nd, *rh, *lh, *rhR;
 6
 7     nd = (Node *) spin_malloc(sizeof(Node));
 8
 9     if (!nd) return FULL;
10
11     nd->R = Dummy;
12     nd->V = v;
13
14     while (true)
15     {   rh = RH;
16         rhR = rh->R;
17         if (rhR == rh)
18         {   nd->L = Dummy;
19             lh = LH;
20             if (DCAS(&RH,&LH,rh,lh,nd,nd))
21                 return OKAY;
22         } else
23         {   nd->L = rh;
24             if (DCAS(&RH,&rh->R,rh,rhR,nd,nd))
25                 return OKAY;
26     }   }
27 }
28
29 val
30 popRight(void)
31 {   Node *rh, *lh, *rhL;
32     val result;
33
34     while (true)
35     {   rh = RH;
36         lh = LH;
37
38         if (rh->R == rh)
39             return EMPTY;
40
41         if (rh == lh)
42         {   if (DCAS(&RH,&LH,rh,lh,Dummy,Dummy))
43                 return rh->V;
44         } else
45         {   rhL = rh->L;
46             if (DCAS(&RH,&rh->L,rh,rhL,rhL,rh))
47             {   result = rh->V;
48                 rh->R = Dummy;
49                 rh->V = null;
50                 return result;
51     }   }   }
52 }
```

summary of a proof of correctness. A few years following its publication an attempt was made to formalize that proof with a theorem prover[22] as part of a master's thesis project.[3] 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.[4] Each proof attempt, for both the original algorithm and the corrected version, reportedly took several months.

Lamport[15] 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[2] 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.[2] Figure 7 reproduces two C routines from Detlefs[2] 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. The

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 indentifies 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 8. C code for a sample test driver.

```
53 void
54 sample_reader(void)
55 {   int i, rv;
56
57     while (!RH)
58     {    /* wait */
59     }
60
61     for (i = 0; i < 10; i++)
62     {   rv = popRight();
63         if (rv != EMPTY)
64         {   assert(rv == i);
65         } else
66         {   i--;
67     }   }
68 }
69
70 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 }
```

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"
%O dcas.c
```

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 *W* and statements executed by the reader process with *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 and retrieves the stored value *0*. The remaining steps in Figure 12 illustrate the execution of the `popRight` routine for that call.

Figure 10. Verification steps.

```
$ time modex -run dcas.c
MODEX Version 2.0 - 2 September 2011
c_code line 111 precondition false:
    (Psample_reader->rv==Psample_reader->i)
wrote model.trail
...
pan: elapsed time 0.02 seconds

7.69 user 4.02 system 0:12.04 elapsed 97% CPU
$
```

Figure 11. Part 1, partial execution of `pushRight` by the test writer.

```
74  W: initialize()
76  W: i = 0
76  W: (i<10)
77  W: # v = pushRight(i) ::
 7  W:   nd = (Node *) spin_malloc(sizeof(Node));
 9  W:   !(!nd)
11  W:   nd->R = Dummy;
12  W:   nd->V = v;
14  W:   (true)
15  W:   rh = RH;
16  W:   rhR = rh->R;
17  W:   (rhR == rh)
18  W:   nd->L = Dummy;
19  W:   lh = LH;
```

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.

Figure 12. Part 2, call to `popRight` by the test reader.

```
57  R: !(!RH)
61  R: i = 0
61  R: (i < 10)
62  R: # rv = popRight() ::
34  R:   (true)
35  R:   rh = RH;
36  R:   lh = LH;
38  R:   !(rh->R == rh)
41  R:   (rh == lh)
42  R:   DCAS(&(RH),&(LH),rh,lh,Dummy,Dummy)
43  R:   return rh->V;
```

Figure 13. Part 3, second call to `popRight` by the reader, with the writer still stalled in its first call to `pushRight`, leading to the assertion violation.

```
62      R: rv = popRight(i)   # rv is 0
63      R: (rv != EMPTY)      # true
64      R: assert(rv == i)    # true
61      R: i++;               # i is now 1
61      R: (i<10)             # true
62      R: rv = popRight()    # rv is again 0
63      R: (rv != EMPTY)      # true
64      R: assert(rv == i)    # false
```

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-

Figure 14. First MSL wheel tracks on Mars.

tection mechanism based on redundancy. An assertion is always meant to be satisfied, meaning that technically its evaluation is almost always redundant. But sometimes the impossible does happen, as when, say, external conditions change in unforeseen ways. Assertions prove their value by detecting off-nominal conditions at the earliest possible point in an execution, thus allowing fault-protection monitors to take action and prevent damage.

The second example of software redundancy was used to protect the critical landing sequence. This was the only phase of the mission in which both the main CPU and its backup were used simultaneously, with the backup in hot standby. Running the same landing software on two CPUs in parallel offers little protection against software defects. Two different versions of the entry-descent-and-landing code were therefore developed, with the version running on the backup CPU a simplified version of the primary version running on the main CPU. In the case where the main CPU would have unexpectedly failed during the landing sequence, the backup CPU was programmed to take control and continue the sequence following the simplified procedure. The backup version of the software was aptly called "second chance," and to everyone's relief proved itself redundant by never being called on to execute.

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

1. Chalin, P. *Ensuring Continued Mainstream Use of Formal Methods: An Assessment*. Roadmap and Issues Group, D.S.R., TR 2005-001, Concordia University, Montréal, Canada, 2005.
2. Detlefs, D.L., Flood, C.H., Garthwaite, A.T. et al. Even better DCAS-based concurrent deques. In *Distributed Algorithms, LNCS Vol. 1914*, M. Herlihy, Ed. Springer Verlag, Heidelberg, 2000, 59–73.
3. Doherty, S. *Modelling and Verifying Non-blocking Algorithms that Use Dynamically Allocated Memory*. Master's Thesis, Victoria University, Wellington, New Zealand, 2004.
4. Doherty, S., Detlefs, D.L., Groves, L. et al. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, P.B. Gibbons and M. Adler, Eds. (Barcelona, Spain, June 27–30). ACM Press, New York, 2004, 216–224.
5. Gluck, P.R. and Holzmann, G.J. Using Spin model checking for flight software verification. In *Proceedings of the 2002 Aerospace Conference* (Big Sky, MT, Mar. 9–16). IEEE Press, Piscataway, NJ, 2002.
6. Havelund, K., Lowry, M., Park, S. et al. Formal analysis of the remote agent: Before and after flight. *IEEE Transactions on Software Engineering 27*, 8 (Aug. 2001), 749–765.
7. Hoare, C.A.R. Assertions: A personal perspective. *IEEE Annals of the History of Computing 25*, 2 (Apr.-June 2003), 14–25.
8. Holzmann, G.J. Scrub: A tool for code reviews. *Innovations in Systems and Software Engineering 6*, 4 (Dec. 2010), 311–318.
9. Holzmann, G.J. The power of ten: Rules for developing safety critical code. *IEEE Computer 39*, 6 (June 2006), 95–97.
10. Holzmann, G.J. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2004.
11. Holzmann, G.J. and Joshi, R. Model-driven software verification. In *Proceedings of the 11th Spin Workshop, LNCS 2989* (Barcelona, Spain, Apr. 1–3). Springer Verlag, Berlin, 2004, 76–91.
12. Holzmann, G.J. and Smith, M.H. Automating software feature verification. *Bell Labs Technical Journal 5*, 2 (Apr.-June 2000), 7–87.
13. Jet Propulsion Laboratory. *JPL Coding Standard for Flight Software*; http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf
14. Kudrjavets, G., Nagappan, N., and Ball, T. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering* (Raleigh, NC, Nov. 7–10). IEEE Press, Piscataway, NJ, 2006, 204–212.
15. Lamport, L. Checking a multithreaded algorithm with +CAL. In *Proceedings of Distributed Computing: 20th International Conference* (Stockholm, Sweden, Sept. 18–20). Springer-Verlag, Berlin, 2006, 151–163.
16. Motor Industry Software Reliability Association. *MISRA-C Guidelines for the Use of the C Language in Critical Systems*. MIRA Ltd., Warwickshire, U.K., 2012; http://www.misra-c.com/
17. NASA. *NASA Engineering and Safety Center, Technical Assessment Report*. National Highway Traffic Safety Administration (NHTSA), Toyota Unintended Acceleration Investigation, Appendix A: Software, Washington, D.C., Jan. 18, 2011; http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA_FR_Appendix_A_Software.pdf
18. Ong, E.C. and Leveson, N. Fault protection in a component-based spacecraft architecture. In *Proceedings of the International Conference on Space Mission Challenges for Information Technology* (Pasadena, CA, July 13–16). Jet Propulsion Laboratory, Pasadena, CA, 2003.
19. Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Providence, RI, Oct. 31–Nov. 1). IEEE Computer Society, Washington, D.C., 1977, 46–57.
20. Redberg, R. and Holzmann, G.J. *Reviewing Code Review*. LaRS Report, Jet Propulsion Laboratory, Pasadena, CA, Nov. 2013.
21. Schneider, F., Easterbrook, S.M., Callahan, J.R., and Holzmann, G.J. Validating requirements for fault-tolerant systems using model checking. In *Proceedings of the International Conference on Requirements Engineering* (Colorado Springs, CO, April 6–10). IEEE Computer Society, Washington, D.C., 1998, 4–13.
22. SRI International, Computer Science Laboratory. *The PVS Specification and Verification System*; http://pvs.csl.sri.com/
23. Vardi, M. and Wolper, P. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science* (Cambridge, MA, June 16–18). IEEE Computer Society, Washington, D.C., 1986, 332–344.

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.