Computer aided verification

lecture 10

Model-checking success stories

Sławomir Lasota
University of Warsaw

LITERATURE

- G. J. Holzman, Mars Code. Commun. ACM 57(2):64-73, 2014.
- D.L. Detlefs, C.H. Flood, A.T. Garthwaite et al. Even better DCAS-based concurrent deques. in *Distributed Algorithms, LNCS Vol. 1914*, 59–73, 2000.
- S. Doherty et al. DCAS is not a silver bullet for nonblocking algorithm design. SPAA 2004: 216-224, 2004.
- T. Ball, V. Levin, S. K. Rajamani, A decade of software model checking with SLAM. Commun. ACM 54(7):68-76, 2011.
- T. Ball, S. K. Rajamani, Bebop: a symbolic model-checker for boolean programs. SPIN Workshop, LNCS 1885, pp 113-130, 2000.

Mars Code

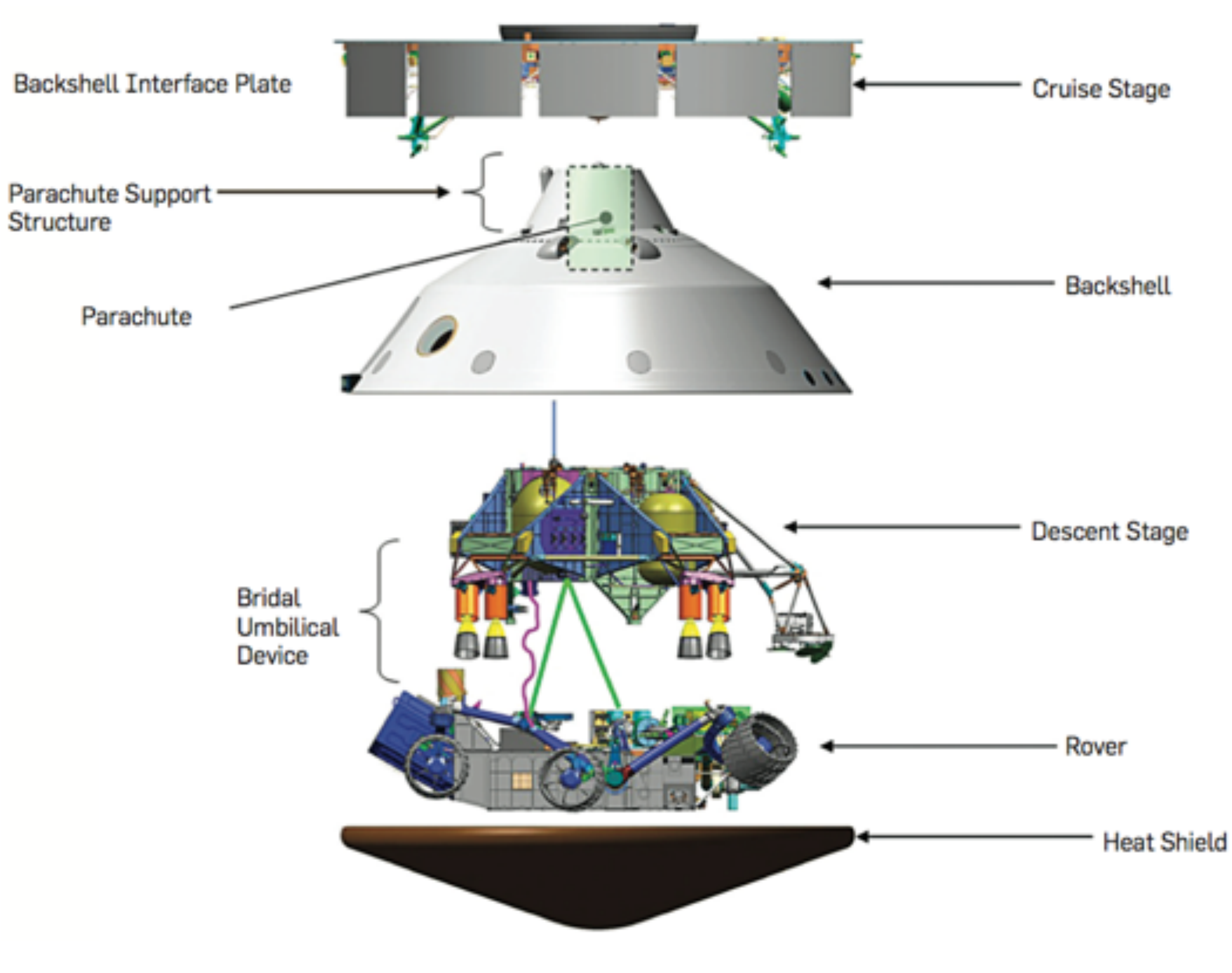
What formal methods were applied by the flight software team in Jet Propulsion Lab, California Institute of Technology (under a contract with NASA), to ensure Curiosity rover reached its destination on Mars on 5 August 2012 (Mars Science Laboratory mission).

OVERVIEW

- 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.
- Extraordinary measures were 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.
- Model checking helped verify intricate software subsystems for the potential of race conditions and deadlocks.

LANDING

The most critical part of the mission



Controlled by one of two computers allocated within the body of the rover.

PRECAUTIONS

Not covered:

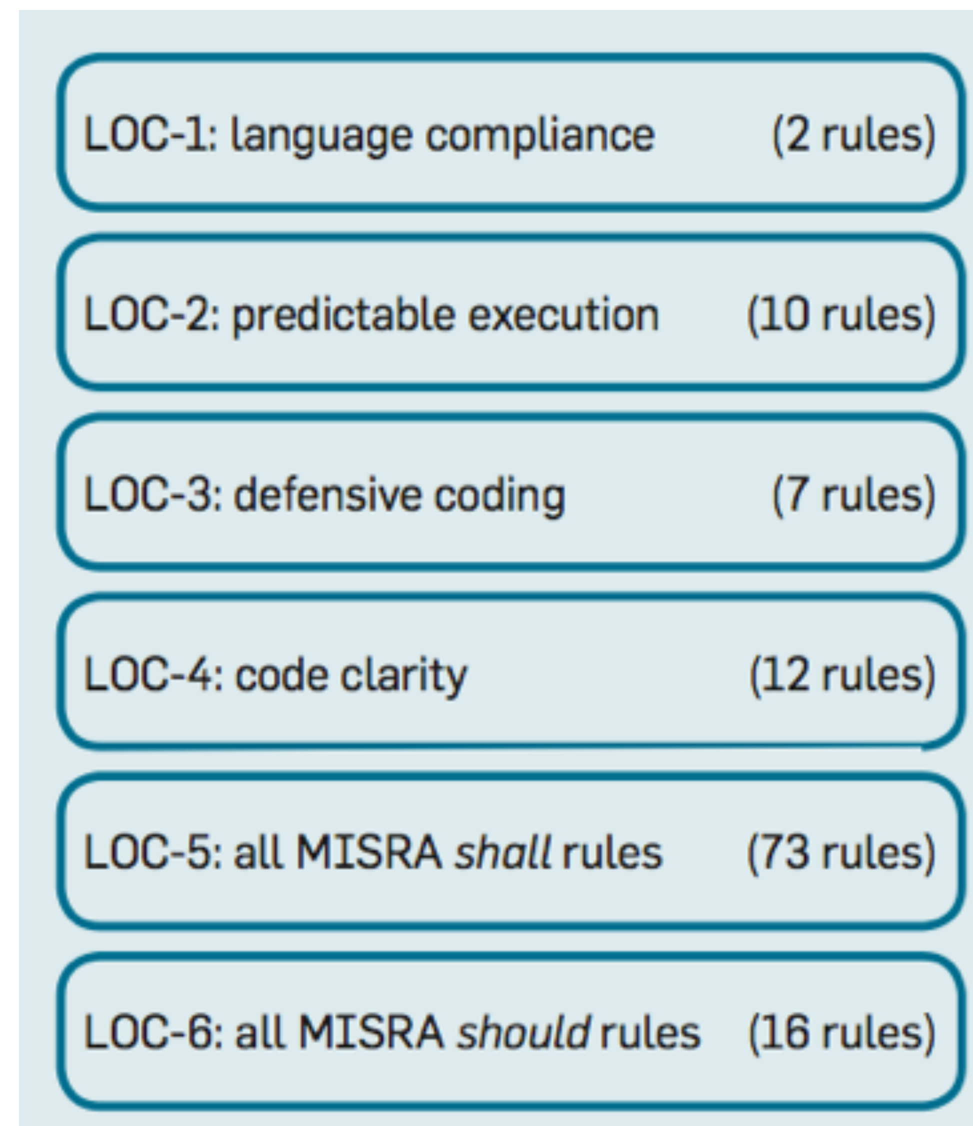
- good software architecture: clean separation of concerns, modularity, strong fault-protection mechanisms, etc.
- good development process: clearly stated requirements, rigorous unit and integration testing, etc.

Covered:

- coding standards adopted
- code review process adopted
- software redundancy
- application of model-checking

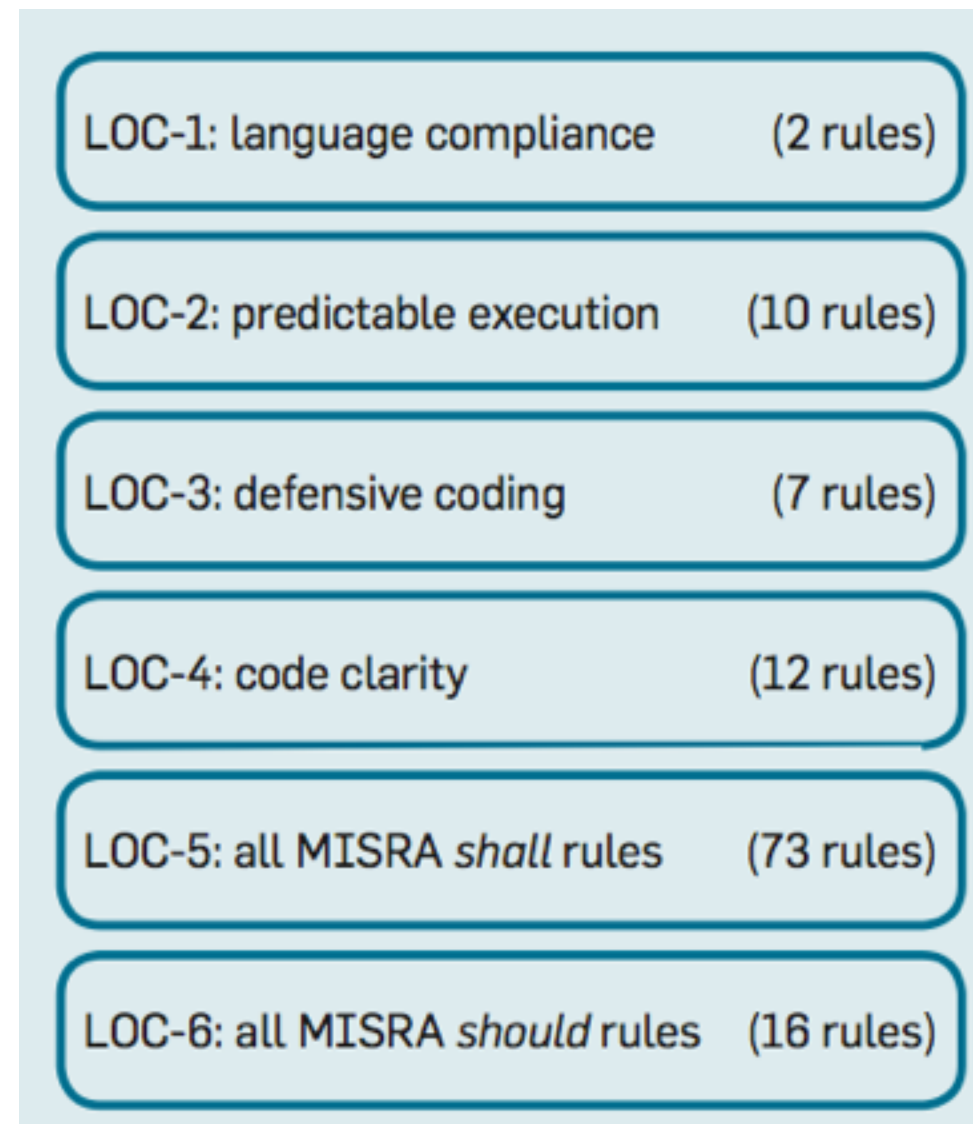
CODING STANDARDS

- risk-related, not style-related, coding rules
- correlate directly with observed risk based on software anomalies from earlier missions
- compliance with a coding rule must be automatically verifiable
- stratified into levels
- automatic compliance check using Coverity, Codesonar and Semmle tools
- flight software developers pass a course (and an exam) on the coding rules



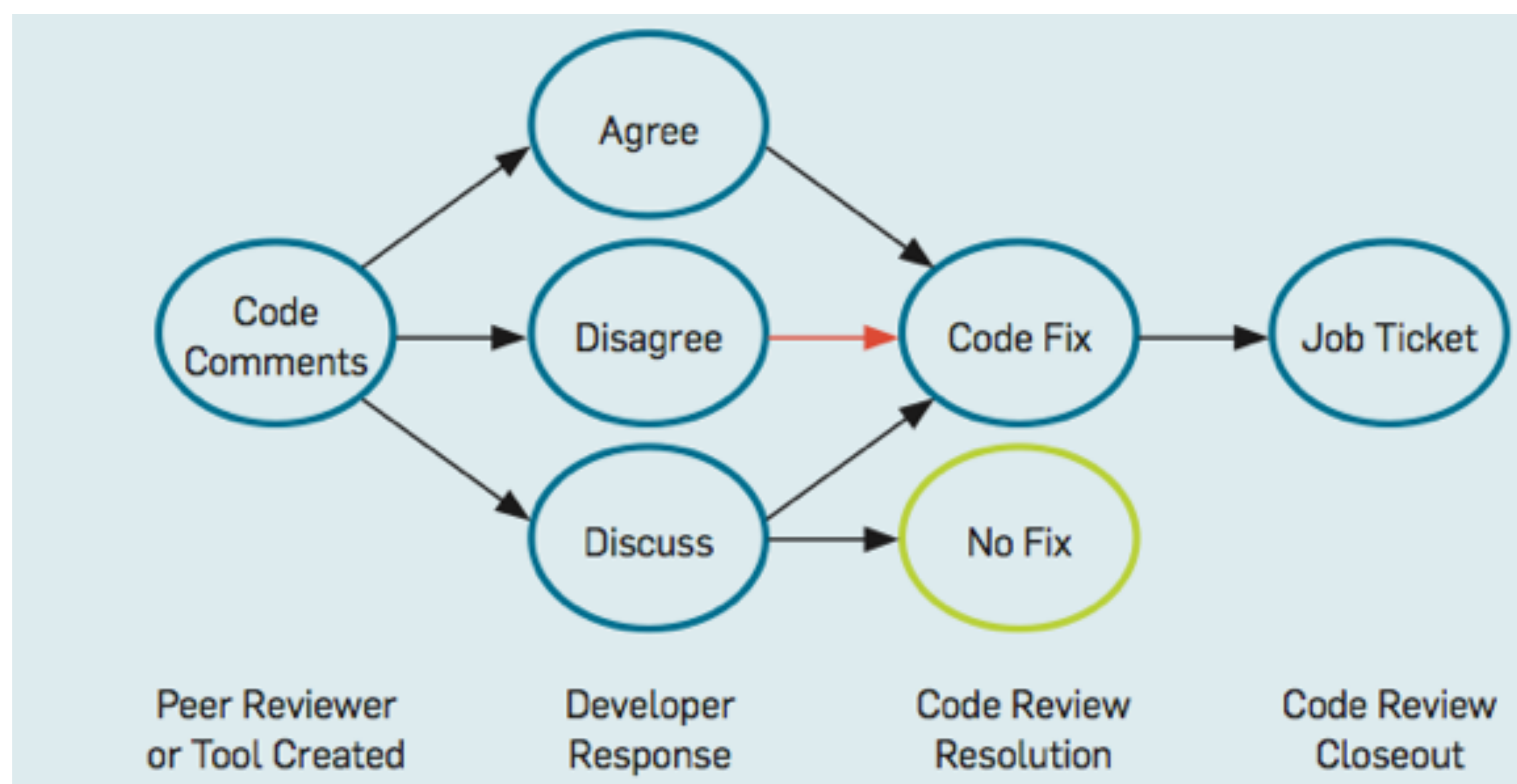
CODING STANDARDS

- LOC-1: compliance with ISO-C99, no compiler or static analyzer warnings
- LOC-2: predictable execution in embedded system context, e.g. loops must have statically verifiable upper bound on the nr of iterations
- LOC-3: e.g. minimal assertion density of 2% (2.26% reached in MSL mission)
- LOC-4 is the target level in mission-critical software, including on-board flight software: restricts use of C preprocessor, function pointers and pointer indirection
- LOC-6 is the target level in safety-critical and human-related software: all rules from the guidelines by Motor Industry Software Reliability Association



CODE REVIEW

- tool-based human reviewers
- simultaneous use of different static analyzers: Coverity, Codesonar, Semmle and Uno to identify likely bugs with reasonable false-positives rate
- designed tool Scrub integrates output of analyzers with human-generated review comments



CODE REVIEW

- 145 code reviews held between 2008 and 2012
- app. 10.000 review comments and 30.000 tool-generated reports discussed
- app. 84% of them led to changes in code (2% difference between human comments and tool-generated reports)
- 12.3% of *disagree* responses of module owners; in 33% a required fix has been done anyway
- 6.4% of *discuss* responses of module owners; in 60% let to changes in code
- critical modules have been reviewed several times

SOFTWARE REDUNDANCY

- critical hardware components were duplicated, including rover's CPU
- on MSL mission all assertions remained enabled during the flight; a failing assertion placed aircraft into a predefined safe state, to diagnose the cause of failure before resuming normal operation
- during the critical landing phase, main CPU and its backup were used simultaneously, running two different versions of controlling software; at failure of main CPU, the backup one was to take control (which did not happen)

MODEL CHECKING

- SPIN previously used in Cassini, Deep Space One and Mars Exploration missions
- MSL mission: 120 parallel tasks under control of real-time operating system, high potential for race conditions
- SPIN + Modex used to verify critical software components:
  - dual-CPU boot-control algorithm
  - the non-volatile flash file system
  - the data-management subsystem (the largest one, 45 k lines of code, converted manually to a Spin model of 1.600 lines)
- model-checking performed routinely after every change in the code of the file system, in most cases identified subtle concurrency flaws

MODEX

Modex builds a SPIN model that consists of three parts:

- user-defined test drivers
- native source-code fragments
- instrumented code fragments, extracted from the source code

DOUBLY LINKED LIST

```
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;
  }
}
```

```
struct Node {valtype V; Node *L; Node *R}

Node *Dummy, *LeftHat, *RightHat;

initially
  Dummy != null and
  Dummy->L == Dummy and Dummy->R == Dummy and
  LeftHat == Dummy and RightHat == Dummy
```

```
pushRight(val v) {
  nd = new Node();
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
    rh = RightHat;
    rhR = rh->R;
    if (rhR == rh) {
      nd->L = Dummy;
      lh = LeftHat;
      if (DCAS(&RightHat, &LeftHat,
                    rh, lh, nd, nd))
        return "ok";
    } else {
      nd->L = rh;
      if (DCAS(&RightHat, &rh->R,
                     rh, rhR, nd, nd))
        return "ok";
    }
  }
}
```

```
val popRight() {
  while (true) {
    rh = RightHat;
    lh = LeftHat;
    if (rh->R == rh) return "empty";
    if (rh == lh) {
      if (DCAS(&RightHat, &LeftHat,
                   rh, lh, Dummy, Dummy))
        return rh->V;
    } else {
      rhL = rh->L;
      if (DCAS(&RightHat, &rh->L,
                    rh, rhL, rhL, rh)) {
        result = rh->V;
        rh->R = Dummy;
        return result;
      }
    }
  }
}
```

MODEX

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
```

Test driver:

```
void
sample_reader(void)
{   int i, rv;

    while (!RH)
    {    /* wait */
    }

    for (i = 0; i < 10; i++)
    {   rv = popRight();
        if (rv != EMPTY)
        {   assert(rv == i);
        } else
        {   i--;
    }   }
}

void
sample_writer(void)
{   int i, v;

    initialize();

    for (i = 0; i < 10; i++)
    {   v = pushRight(i);
        if (v != OKAY)
        {   i--;
    }   }
}
```

popRight returns "empty" even if queue is never empty

- A process p invokes **popRight** while the deque is not empty. It loads its **rh** variable and is then delayed.
- While p is delayed, other processes complete **pushRight** and **popLeft** operations so that the node referenced by p's **rh** variable is popped from the deque by a **popLeft** *without the deque being empty in that period.*
- p resumes execution and performs the test at line 5, finding **rh**→**R** = **rh** (because **rh** has been removed by a **popLeft**), and returns **empty**.

```
pushRight(val v) {
  nd = new Node();
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
    rh = RightHat;
    rhR = rh->R;
    if (rhR == rh) {
      nd->L = Dummy;
      lh = LeftHat;
      if (DCAS(&RightHat, &LeftHat,
               rh, lh, nd, nd))
        return "ok";
    } else {
      nd->L = rh;
      if (DCAS(&RightHat, &rh->R,
               rh, rhR, nd, nd))
        return "ok";
    }
  }
}
```

```
1  val popRight() {
2    while (true) {
3      rh = RightHat;
4      lh = LeftHat;
5      if (rh->R == rh) return "empty";
6      if (rh == lh) {
7        if (DCAS(&RightHat, &LeftHat,
8                 rh, lh, Dummy, Dummy))
9          return rh->V;
10     } else {
11       rhL = rh->L;
12       if (DCAS(&RightHat, &rh->L,
13                rh, rhL, rhL, rh)) {
14         result = rh->V;
15         rh->R = Dummy;
16         return result;
17       }
18     }
19   }
20 }
```

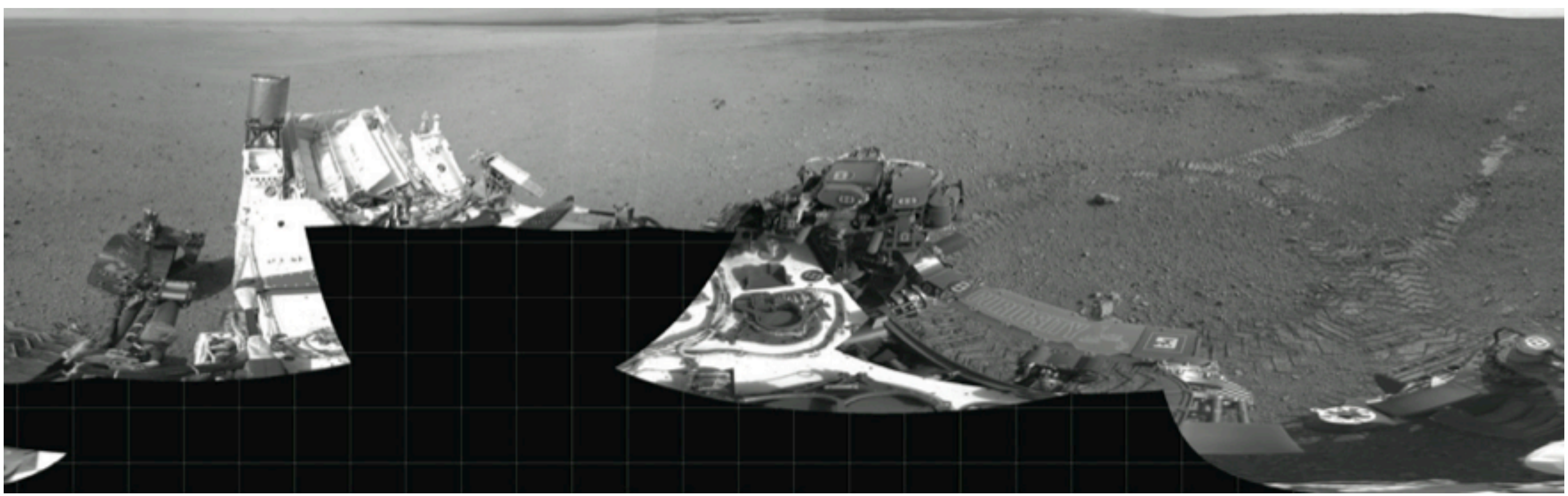
popRight returns the same element twice

- Process p invokes **popRight** when the deque contains more than one element and runs alone until it is about to execute the DCAS at line 12, but is delayed before it does so.
- Other processes execute **pushRight** and **popLeft** operations so that p.rh = LeftHat and the deque contains more than one element. This can be achieved without modifying p.rh→L.
- Some process q invokes and completes an execution of **popLeft**, and this operation removes the node referenced by p.rh. This also happens without modifying p.rh→L.
- Other processes execute **popRight** operations so that once again, p.rh = RightHat. The deque is now empty. Finally, p executes its DCAS, which succeeds because p.rh = RightHat and p.rh→L = p.rhL, and p returns p.rh→V, which has already been returned by q.

```
pushRight(val v) {
  nd = new Node();
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
    rh = RightHat;
    rhR = rh->R;
    if (rhR == rh) {
      nd->L = Dummy;
      lh = LeftHat;
      if (DCAS(&RightHat, &LeftHat,
               rh, lh, nd, nd))
        return "ok";
    } else {
      nd->L = rh;
      if (DCAS(&RightHat, &rh->R,
                rh, rhR, nd, nd))
        return "ok";
    }
  }
}
```

```
1  val popRight() {
2    while (true) {
3      rh = RightHat;
4      lh = LeftHat;
5      if (rh->R == rh) return "empty";
6      if (rh == lh) {
7        if (DCAS(&RightHat, &LeftHat,
8                 rh, lh, Dummy, Dummy))
9          return rh->V;
10     } else {
11       rhL = rh->L;
12       if (DCAS(&RightHat, &rh->L,
13                rh, rhL, rhL, rh)) {
14         result = rh->V;
15         rh->R = Dummy;
16         return result;
17       }
18     }
19   }
20 }
```

First MSL wheel tracks on Mars:



SLAM

SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- one of reasons is the complexity of the Windows drivers API
- SLAM: automatically checks device drivers for certain correctness properties with respect to the Windows device drivers API
- now core of Static Driver Verifier, which in turn is a part of Windows Driver Development Kit, a toolset for drivers developers, and integrated into Visual Studio

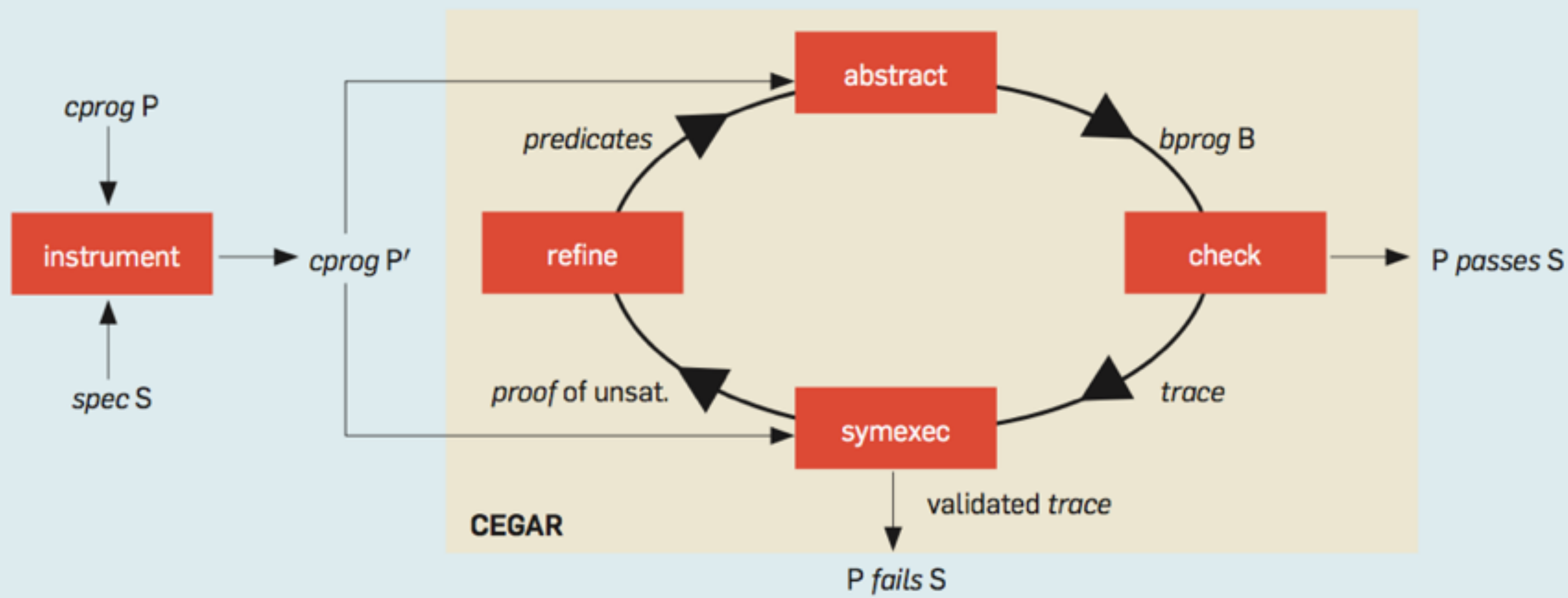
SOFTWARE SUCCESS STORY

- 85% of system crashes of Windows XP caused by bugs in third-party kernel-level device drivers (2003)
- one of reasons is the complexity of the Windows drivers API
- SLAM: automatically checks device drivers for certain correctness properties with respect to the Windows device drivers API
- now core of Static Driver Verifier, which in turn is a part of Windows Driver Development Kit, a toolset for drivers developers, and integrated into Visual Studio

TECHNIQUES

- abstracts C programs into boolean programs and applies a abstraction refinement scheme (CEGAR)
- recursive (!) procedure calls (pushdown systems model-checking)
- symbolic model checking (BDDs)
- pointers (pointer-alias static analysis)
- principal application: checking whether device drivers satisfy driver API usage rules
- API rules specified in SLIC (Specification Language for Interface Checking)
- temporal safety properties

CEGAR




```
type cprog, spec, predicates, bprog, trace, proof

type result =
   Pass | Fail of trace

type chkresult =
   AbstractPass | AbstractFail of trace

type excresult =
   Satisable | Unsatisable of proof
```

```
let rec cegar (P':cprog) (preds :predicates) : result =
   let B: bprog = abstract (P',preds) in
   match check(B) with
   | AbstractPass -> Pass
   | AbstractFail(trc) ->
     match symexec(P', trc) with
     | Satisable -> Fail(trc)
     | Unsatisable(prf) -> cegar P' ( preds ∪ (refine prf))

let slam ( P:cprog) (S:spec) : result =
   cegar (instrument (P,S)) (preds S)
```

SLIC

- SLIC rule is essentially a safety automaton defined in C-like language that monitors a program's execution at function calls and returns
- only reads program variables
- can maintain information about history
- signals occurrence of a bad state
- SLIC rule consists of
  - state variables
  - event handlers
  - binders to event in the code (not shown)

```
state { enum {Unlocked, Locked} state; }

KeInitializeSpinLock.call {
    state = Unlocked;
}

KeAcquireSpinLock.call {
    if ( state == Locked ) {
        error;
    } else {
        state = Locked;
    }
}

KeReleaseSpinLock.call {
    if ( !(state == Locked) ) {
        error;
    } else {
        state = Unlocked;
    }
}
```

CODE INSTRUMENTATION

```
state { enum {Unlocked, Locked} state; }

KeInitializeSpinLock.call {
    state = Unlocked;
}

KeAcquireSpinLock.call {
    if ( state == Locked ) {
        error;
    } else {
        state = Locked;
    }
}

KeReleaseSpinLock.call {
    if ( !(state == Locked) ) {
        error;
    } else {
        state = Unlocked;
    }
}
```

```
..
KeInitializeSpinLock();
..
..
if(x > 0)
  KeAcquireSpinlock();
count = count+1;
devicebuffer[count] = localbuffer[count];
if(x > 0)
  KeReleaseSpinLock();
...
...
```

```
..
{ state = Unlocked;
  KeInitializeSpinLock();}
..
..
if(x > 0)
  { SLIC_KeAcquireSpinLock_call();
    KeAcquireSpinlock(); }
count = count+1;
devicebuffer[count] = localbuffer[count];
if(x > 0)
  { SLIC_KeReleaseSpinLock_call();
    KeReleaseSpinLock(); }
...
...
```

CEGAR AT WORK

```
slic_error() { assert(false); }

bool {state==Locked};

SLIC_KeAcquireSpinLock_call() {
  if( {state==Locked}) slic_error();
  else {state==Locked} := true;
}

SLIC_KeReleaseSpinLock_call() {
  if( !{state==Locked}) slic_error();
  else {state==Locked} := false;
}
```

```
...
...
{state==Locked} := false;
KeInitializeSpinLock();
...
...
if(*)
  { SLIC_KeAcquireSpinLock_call();
     KeAcquireSpinLock(); }
skip;
skip;
if(*)
  {  SLIC_KeReleaseSpinLock_Call();
     KeReleaseSpinLock(); }
...
...
```

```
bool {x > 0};
...
{state==Locked} := false;
KeInitializeSpinLock();
...
...
if({x>0})
  {  SLIC_KeAcquireSpinLock_call();
     KeAcquireSpinLock(); }
skip;
skip;
if({x>0})
  {  SLIC_KeReleaseSpinLock_Call();
     KeReleaseSpinLock(); }
..
...
```

FROM SLAM TO SDV

- fully automatic ("push-button technology"): SDV wraps SLAM with scripts, input-output routines, API rules, environment model, etc.
- pre-defined API rules, written by SDV team; different rules for different classes of APIs
- verifies source code of a device driver agains a SLIC rule
- code of a device driver is sandwiched between:
    - top layer "harness" (test drive): main routine that calls driver entry points
    - bottom layer: stub for Windows API functions (overapproximation), which define "environment" model
- dynamic memory allocation in preprocessing in harness

API RULES

- different requirements for different classes of APIs, for instance:
    - NDIS API for network drivers
    - MPIO API for storage drivers
    - WDM API for display drivers
- WDF API - high level abstraction for common device drivers
- WDF API rules influenced WDF design, to make it easier to verify !
- version 2.0 of SDV (Windows 7, 2009) comes with >210 API rules for WDM, WDF and NDIS APIs

WHO WRITES API RULES ?

- typical end-user does not write API rules
- initially written and iteratively refined in cooperation with driver experts (“It takes a PhD to develop API rules”)
- since 2007 task of writing API rules transferred to software engineers
- in version 2.0 of SDV (Windows 7, 2009), out of >210 API rules:
  - 60 written by formal verification experts
  - 150 written or adapted by software engineers or interns

EFFECTIVENESS

- SDV 1.3: on average 1 bug per driver in 30 sample drivers shipped with Driver Development Kit for Windows Server 2003.
- SDV 1.4, 1.5 (Windows Vista drivers): on average 1 bug per 2 drivers in sample WDM drivers
- SDV 1.6: on average 1 bug per 3 drivers in sample WDF drivers for Windows Server 2008
- SDV 2.0: on average 1 bug per WDF driver, and few bugs in all WDM sample drivers
  - on WDM drivers: 90% real bugs, 10% false alarms, 3.5% nonresults
  - on WDF drivers: 98% real bugs, 2% false alarms, 0.04% nonresults
  - during development of Windows 7, 270 real bugs found in 140 WDM and WDF drivers

PERFORMANCE

- a run of SDV on 100 drivers and 80 SLIC rules:
    - largest driver: 30 k lines of code
    - total of all drivers: 450 k lines of code
    - total time of 8.000 runs: 30 hours on 8-core machine
    - timeout: SDV run is killed after 20 min
    - results in 97% of runs

LIMITATIONS OF SLAM

- unable to handle large programs
- often gives useful result for control-dominated properties of programs with tens of thousands lines of code
- unable to establish properties that depend on heap data structure (sound overapproximation of pointers)
- no support for concurrent programs (there is however an extension towards concurrent programs: context-bounded analysis of pushdown systems)

Model-checking pushdown systems

Reachability for pushdown systems

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S
- Configurations of B: $Q \times S^*$

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S
- Configurations of B: $Q \times S^*$
- finite automaton A with states Q and input alphabet S

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S
- Configurations of B: $Q \times S^*$
- finite automaton A with states Q and input alphabet S
- L(A) = { (q, w) : A accepts w from state q }

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S
- Configurations of B: $Q \times S^*$
- finite automaton A with states Q and input alphabet S
- L(A) = { (q, w) : A accepts w from state q }

Theorem: Pre*(regular set) is regular,
and may be effectively computed in polynomial time

Reachability for pushdown systems

- pushdown system B, with states Q and stack alphabet S
- Configurations of B: $Q \times S^*$
- finite automaton A with states Q and input alphabet S
- L(A) = { (q, w) : A accepts w from state q }

Theorem: Pre*(regular set) is regular,
and may be effectively computed in polynomial time

Corollary: Configuration-to-configuration reachability is decidable
in polynomial time

Idea

Saturate transitions $\delta \subseteq Q \times S \times Q$ of automaton A:

$\delta' := \delta \cup pop$

repeat

$\delta' := \delta \cup forced(\delta')$

until $forced(\delta') \subseteq \delta'$

Outcome: $\delta'(p, s, q)$ in A iff $(p, s) \rightarrow^* (q, \varepsilon)$ in B

$(p, s, q) \in forced(\delta')$ iff
PDA B has a push transition
$(p, s, q_2, s_2 s_1)$ such that
$(q_2, s_2, q_1), (q_1, s_1, q) \in \delta'$