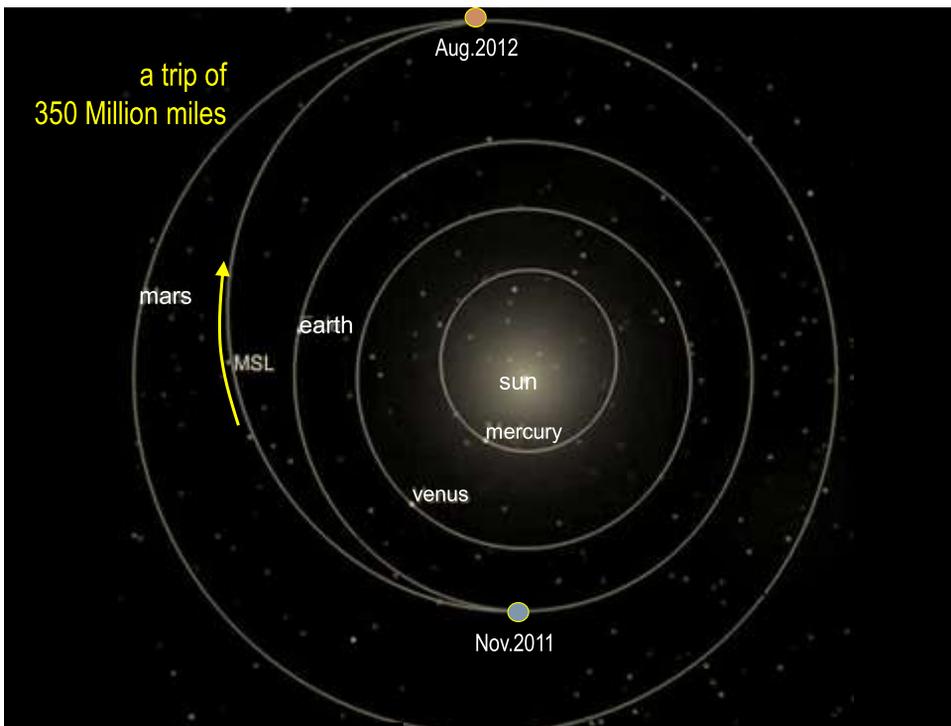


Proof or Consequences the Verification of Curiosity's Software

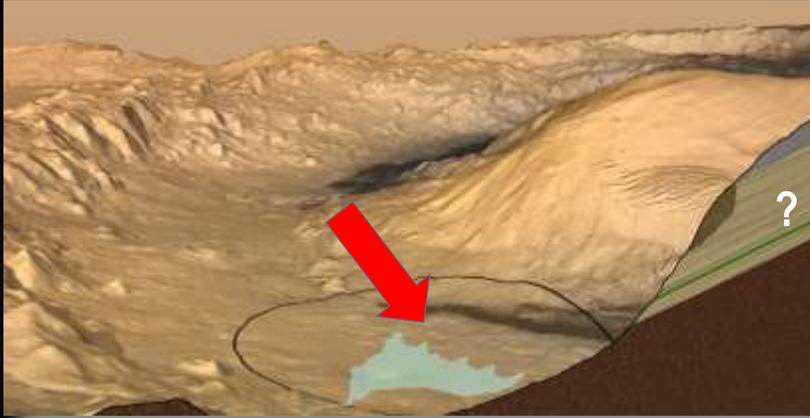
Gerard J. Holzmann
JPL Laboratory for Reliable Software
Caltech CMS, Senior Faculty Associate

© 2012 California Institute of Technology.
Government Sponsorship Acknowledged.

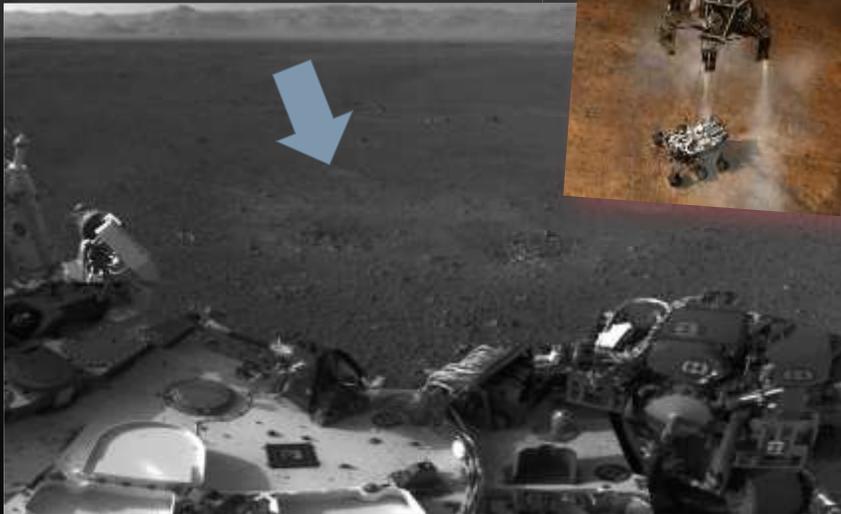


target: *Gale* crater

an old streambed



12 x 4.3 mile landing ellipse

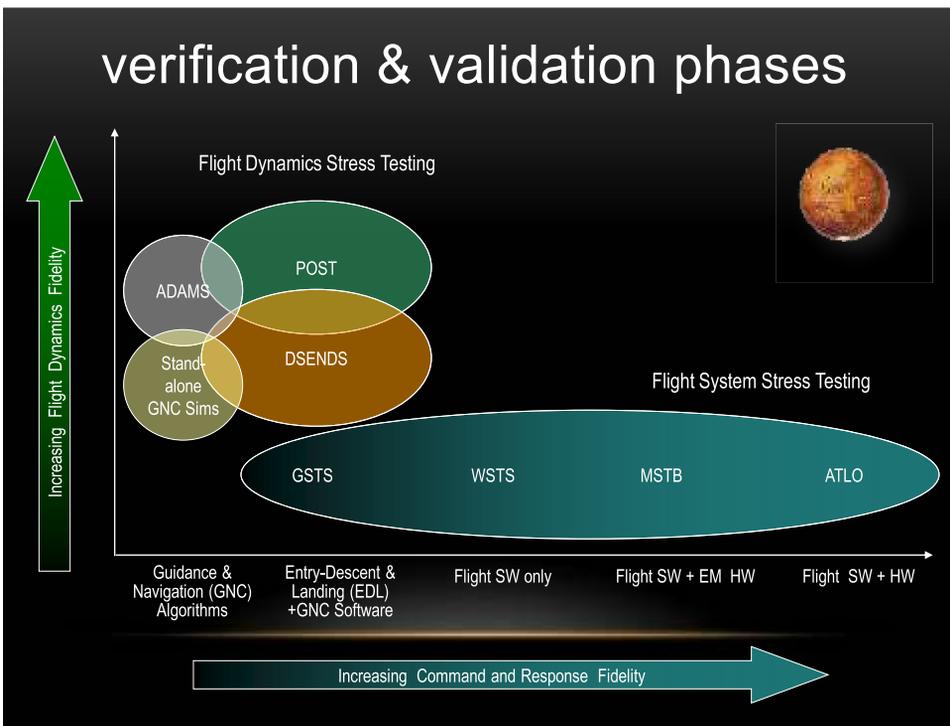


surface blast marks from the landing

how do you make sure that it works?



verification & validation phases



the flight software...

3 Million lines of code

120 parallel threads ('tasks')

5 years development time

how do you get it right?

- new risk-based coding standard
 - with nightly compliance checks
- new developer certification program
 - with exams...
- new code review process
 - based on static source code analysis
- **formal analysis of critical sub-systems**

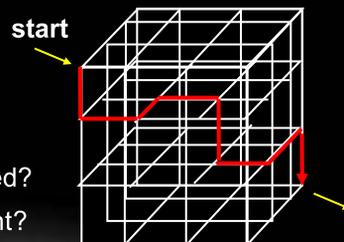
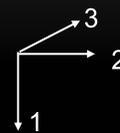
first challenge: multi-threaded code

- what is the number of possible executions given 3 processes with 3 interleaving points in each?

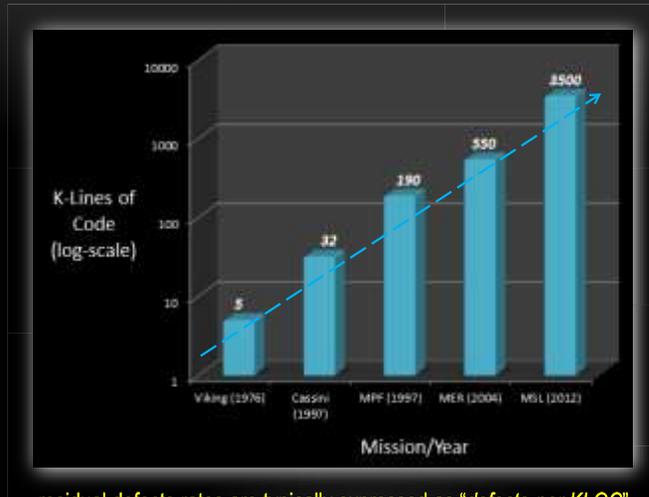
$$\frac{9!}{6! \cdot 3!} \cdot \frac{6!}{3! \cdot 3!} \cdot \frac{3!}{3!} = 1,680$$

(placing 3 sets of 3 tokens in 9 slots)

- are all these executions okay?
 - in tests, how many are checked?
 - how many paths are equivalent?



second challenge: growing code size



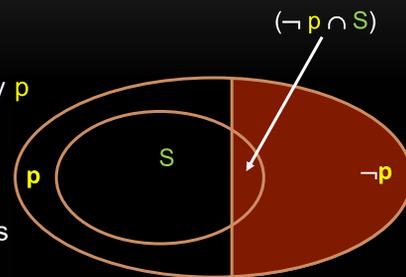
a new meaning for
"exhaustive testing"?
(exhausting the testers,
not the code)

residual defects rates are typically expressed as "defects per KLOC"
a rate of 0.1 – 1.0 defects per KLOC is considered difficult to achieve

9

a more formal approach *model checking*

- the basic idea:
given formal system S and property p
negate p and compute: $\neg p \cap S$
- for multi-threaded code:
 S is computed from thread behaviors
(using partial order reduction to
eliminate equivalent executions)
- p is expressed in linear temporal logic
which defines the accepting language



if $(\neg p \times S)$ is empty: p holds in S

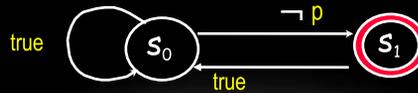
if non-empty: $(\neg p \times S)$ contains
all error sequences in S
compute one such witness and stop

the relation between LTL and automata

- for any LTL formula f there exists a finite, non-deterministic, Büchi automaton that accepts exactly (and only) those executions (ω -runs) for which f is satisfied
 - example $f: \diamond \square p$ corresponds to Büchi automaton:



- the logical negation of $f: \neg \diamond \square p$



we can use this to check multi-threaded code (like key parts of Curiosity's flight software)

7

proving correctness of multi-threaded C programs – what is the state-of-the-art?

- a small example

2000: manual proof (a few months)
proof sketch: 5 pages, 7 Lemmas, 5 Theorems

2004: PVS theorem prover (3 months)

2006: +CAL model & TLA+ proof (a few days)

2012: ?

Erin Dubler: DCAS-Based Concurrent Design

2000

DCAS is not a Silver Bullet for Nonblocking Algorithm Design

2004

Checking a Multithreaded Algorithm with +CAL

2006

Lestle Lamport

Microsoft Research

11/01/2006

To appear in DRS' 2006

Abstract

Let us check a multithreaded algorithm that was written in a language that implements the algorithm in the "C" algorithm. The TLA+ model checker on it, and found the error. The algorithm will have to be corrected, why should algorithm designers believe model checking?

12

the DCAS algorithm in C

(from [Detlefs et al 2000])

```
int
DCAS(Node **addr1, Node **addr2,
      Node *old1,  Node *old2,
      Node *new1,  Node *new2)
{
    /* atomically */
    if ((*addr1 == old1) && (*addr2 == old2))
    {
        *addr1 = new1;
        *addr2 = new2;
        return 1; /* true */
    }
    /* else */
    return 0; /* false */
}
```

semantics of the DCAS operation

the proposed pushRight() and popRight() code

```
Node *Dummy, *LeftHat, *RightHat;
void
pushRight(val v)
{
    Node *nl, *rh, *lh, *rh0;
    nl = (Node *) dcas_malloc(sizeof(Node));
    if (!nl) return FAIL;
    nd->r = Dummy;
    nd->l = nl;
    while (true) {
        rh = RightHat;
        rh0 = rh->r;
        if (rh0 == rh) {
            rh->l = Dummy;
            lh = LeftHat;
        } else {
            if (DCAS@RightHat, &LeftHat, rh, lh, nl, nl)
                return OKAY;
        }
    }
}

void
popRight(void)
{
    Node *nl, *rh, *rh0;
    val result;
    while (true) {
        rh = RightHat;
        lh = LeftHat;
        if (rh->r == rh)
            return DUFFY;
        if (rh == rh0) {
            if (DCAS@RightHat, &LeftHat, rh, lh, Dummy, Dummy)
                return DUFFY;
        } else {
            rh0 = rh->r;
            if (DCAS@RightHat, &rh0, rh, rh0, rh0, rh0) {
                result = rh->l;
                rh->l = Dummy;
                rh->r = nl;
                return result;
            }
        }
    }
}
```

13

a simple tester

```
int dcas_cnt;
char dcas_heap[MAX_HEAP];
char *
dcas_malloc(unsigned int n)
{
    char *p = (char *) 0;
    if (dcas_cnt + n < sizeof(dcas_heap))
    {
        p = &dcas_heap[dcas_cnt];
        dcas_cnt += n;
    }
    return p;
}
```

an explicit heap allocator

```
$ ncs1 dcas.c
sloc   ncs1  comments  file
180    146          6  dcas.c
20     16           0  dcas.h
```

a sample reader and writer thread

```
void
initialize()
{
    Dummy->l = Dummy->r = Dummy;
    LeftHat = Dummy;
    RightHat = Dummy;
}

void
sample_reader(void)
{
    int i, rv;
    while (!RightHat)
        /* wait */
    for (i = 0; i < 10; i++)
        rv = popRight();
    if (rv != DUFFY)
        assert(rv == 1);
    else
        i--;
}

void
sample_writer(void)
{
    int i, v;
    initialize();
    for (i = 0; i < 10; i++)
        v = pushRight(1);
    if (v != OKAY)
        i--;
}
```

verification

```
# wouldn't it be nice
# if we could just do this:
$ verify dcas.c
..report assertion violation
$
```

we can!

1. this takes the **C code** as input and uses a model-extractor to generate a formal model (**S**); it then runs the Spin model to check if the assertion (**p**) can be violated
2. all steps together take **10 seconds**
3. the verification step takes a fraction of a second

```
total 12
-rw-r--r-x 1 gh gh 2748 Oct 16 20:24 dcas.c
-rw-r--r-x 1 gh gh 321 Oct 16 20:09 dcas.h
-rw-r--r-x 1 gh gh 147 Oct 16 20:23 dcas.prx
$ modex -run dcas.c
MODEX Version 2.0 - 2 October 2012
./modex -run dcas
pan:11 c_code line 98 precondition false: (sample_reader->rc==sample_reader->rc) (at depth 1749)
pan: wrote model.trail
(Cspin Version 6.2.1 -- 29 September 2002)
Warning: search not completed
+ Partial Order Reduction
Hash-Compact 4 search for:
never claim - (none specified)
SAFETY)
: 1
s Stored (State-vector + overhead)
compression: 16.50%
+ 38 byte overhead
0.194 memory used for DFS stack (-m10200)
4097.315 total actual memory usage
pan: elapsed time 0.01 seconds
```

model extraction

```
$ ls -l dcas.*
-rw-r--r-x 1 gh Name 2764 Oct 17 11:22 dcas.c
-rw-r--r-x 1 gh Name 321 Oct 17 11:09 dcas.h
-rw-r--r-x 1 gh Name 147 Oct 17 11:09 dcas.prx
$ ncs1 dcas.*
sloc  ccol  comments  file
189   146      0  dcas.c
20    16      0  dcas.h
10    10      0  dcas.prx
219   172      6  total
```

the code: 200 lines

the secret: 10 lines

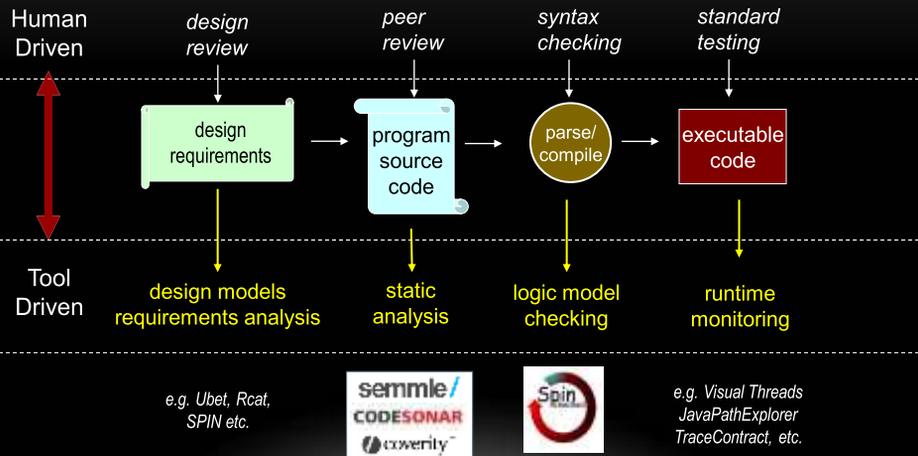
```
%F dcas.c
%X -e pushRight
%X -e popRight
%X -e initialize
%X -a sample_reader
%X -a sample_writer
%D
#include "dcas.h"
%O dcas.c
```

a small configuration file
to guide the modex model extractor

*dcas.c contains the test routines
and the explicit heap allocator,
which makes sure all relevant data
is tracked as part of the system state*

*the model extractor preserves control-flow
it supports event-level **abstractions** (not shown)*

where else can we use this? model based testing



17

the journey begins...

