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.

a trip of
350 Million miles

Aug.2012

Nov.2011
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?

\[
\begin{align*}
9! & \ 
6! & \ 
3! & \\
\frac{6!}{3!} & \ 
\frac{3!}{3!} & \ 
3! & = 1,680
\end{align*}
\]

(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

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.

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

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, B"uchi automaton that accepts exactly (and only) those executions ($\omega$-runs) for which $f$ is satisfied
  - example $f$: $\Diamond \Box p$ corresponds to B"uchi automaton:

- the logical negation of $f$: $\neg \Diamond \Box p$

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

---

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: ?
the DCAS algorithm in C
(from [Detlefs et al 2000])

```c
DCAS(Mode **add1, Mode **add2, Mode *old1, Mode *old2)
{
    if ((add1 == old1) && (*add2 == old2))
        *add1 = *add2;
    else
        return 1; /* true */
    else
        return 0; /* false */
}
```

semantics of the DCAS operation

```c

```

the proposed pushRight() and popRight() code

```

a simple tester

```c

```

an explicit heap allocator

```c

```

da sample reader and writer thread
verification

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

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

model extraction

$ lcs -I dcas
$ ncs dcas

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 code: 200 lines
the secret: 10 lines

a small configuration file
to guide the modex model extractor
the model extractor preserves control-flow
it supports event-level abstractions (not shown)
where else can we use this?
model based testing

Human Driven
- design review
  - design requirements
  - program source code
- peer review
  - parse/compile
  - executable code
- syntax checking
- standard testing

Tool Driven
- design models requirements analysis
- static analysis
- logic model checking
- runtime monitoring

e.g. Umet, Roat, SPIN etc.

the journey begins…