Bug Chasing by Formal Verification

Gérard Berry
Collège de France
Chair Algorithms, Machines, and Languages
Académie des sciences, Académie des technologies

gerard.berry@college-de-france.fr
http://www.college-de-france.fr/site/gerard-berry/index.htm

Prague, October 15, 2015
The Original Computer Bug

Grace Hopper, 9 septembre 1947, 15h45
A Wonderful Time Bug

• Lecteur MP3 / video Zune, 1ᵉ janvier 2008

```c
year = ORIGINYEAR; /* = 1980 */
while (days > 365)
{
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    } else {
        days -= 365;
        year += 1;
    }
}

```

Damned!

Fix: remove the battery or wait for 24h

See also: Sony PS3 Fat bug, iPhone's alarm clock bug, etc.
A Very Sad Time Bug

• Dharan, Feb. 25th, 1991, Patriot missile bug
  – 1/10 s rounding rapidly degrades time-of-day quality
  – after 110h, error = 0.34 s
  – the Patriot misses the Scud
  – 28 dead soldiers, 98 wounded

Fix: reboot the system every few hours
Buffer Overflow

• A program that fails one day a week
Buffer Overflow

- A program that fails one day a week
Buffer Overflows

- Lots of buffer overflows for strings in navigators etc.
- Buffer overflows in peripheral drivers are **killers**

Major source of headache for Microsoft and others
Major entry points for malicious programs

Major incentive for using formal verification
Hardware Bugs

• Pentium, 1994: floating-point division bug

\[ \frac{4195835}{3145727} = 1.33820449136241002 \]

Official cost: $475,000,000!

• AMD Phenom cache deadlock, 2008

• Intel Sandy Bridge bug 2011
  One misdimensioned transistor ⇒ $1,000,000,000!
A Reference Spatial Bug

• Ariane 501 explosion, June 4th, 1996
  – unprotected overflow in a fp32 → int16 conversion
  – Both gyrolasers declare failure → loss of control
  – This code was useless, everything was working perfect!
  – Ariane 5 was simulated on Ariane 4's trajectory, etc.
Martian Space Bugs

• Pathfinder near-loss from task priority inversion, 1997
  – the bug could be identified by running a copy on earth
  – it could be fixed by remote-patching the system!
  – the solution was already published, but ignored...

• Spirit rover endless reboot
  – software problem within flash memory file system
  – remote-patched by excluding flash memory from reboot

• Mars Orbiter crash
  – confusion between metric and US units...
Medical Bugs

• Therac 25 patient irradiation, 1985-1987
  – massive overdose resulting in death and serious injuries
  – mechanical protection replaced by software protection
  – subtle series of bugs, some related to keyboard handling
  – poor error management, bad fix (remove UP key!), etc.
  – see paper by N. Levenson and C. Turner

• Pacemaker security flaw (U. Washington and Mass.)
  – get wireless access to a heart defibrillator and pacemaker
  – makes it possible to shut down, send shocks, or get patient data
Yet to come: coordination with other cars and the town
Report on Toyota's Engine Control Software

There are a large number of functions that are overly complex. By the standard industry metrics some of them are untestable, meaning that it is so complicated a recipe that there is no way to develop a reliable test suite or test methodology to test all the possible things that can happen in it. Some of them are even so complex that they are what is called unmaintainable, which means that if you go in to fix a bug or to make a change, you're likely to create a new bug in the process. Just because your car has the latest version of the firmware -- that is what we call embedded software -- doesn't mean it is safer necessarily than the older one….And that conclusion is that the failsafes are inadequate. The failsafes that they have contain defects or gaps. But on the whole, the safety architecture is a house of cards. It is possible for a large percentage of the failsafes to be disabled at the same time that the throttle control is lost.

Michael Barr, American justice expert
(800 pages report)
Distributed Algorithms Bugs

• AT&T long distance telephone crash, 1990
  – misplaced break statement in C code
  – change was not tested before deployment

• Facebook outage, Sept. 2010
  – self denial of service attack due to bad database manipulation
Simulation Problems (Simulink)

\[
x_1 : \textbf{init } d_1 \\
x_1 = v_1 \\
x_2 : \textbf{init } d_2 \\
x_2 = v_2 \\
v_1 : \textbf{init } v \text{ every } [x_1 \textbf{ up } x_2 \Rightarrow last(v_2)] \\
v_2 : \textbf{init } 0 \text{ every } [x_1 \textbf{ up } x_2 \Rightarrow last(v_1), x_2 \textbf{ up } d_3 \Rightarrow -last(v_2)] \\
v_1 = v_2 = 0
\]

schoks ⇒ actions
Simulation Problems (Simulink)

\[ x_1 \quad \text{v} \quad x_2 \]

\[ d_1 \quad d_2 = d_3 \]
Simulation Problems (Simulink)

\[ d_1 \quad d_2 = d_3 \]
Simulation Problems (Simulink)
Simulation Problems (Simulink)

Zélus: Pouzet et. al., ENS / Inria
Simulation Problems (Simulink)

\[ d_1 \quad \text{and} \quad d_2 = d_3 \]

Zélus: Pouzet et al., ENS / Inria
Man / Computer, a Big Chasm to Fill

Intuitive
Rigorous
Slow

Mastering?

Hyper-fast
Hyper-exact
Hyper-stupid
TDGGTDTDTGGDGGDGDTGDTGDGT
TDGGTDTDGDDTDGDTGDGT
TDGGTDTTTGDDTDGDGT
TDGGTDTDGDDTDGDGT
TDGGTDTTTGTDGDGT
Deadlock

Lise and Laure
Starvation

Lise, Laure, and Manon
Skiing in the 20th Century
Skiing in the 20th Century
Skiing in the 20th Century

13:00
Skiing in the 20\textsuperscript{th} Century
Skiing in the 20th Century
Skiing in the 20th Century, With a Protocol
Skiing in the 20th Century, With a Protocol
Skiing in the 20th Century, With a Protocol
Skiing in the 20th Century, With a Protocol
Skiing in the 20th Century, With a Protocol

13:06
Skiing in the 20th Century, With a Protocol
Skiing in the 20th Century, With a Protocol

13:06

The flag adds the missing causality link
Skiing in the 21st Century
Skiing in the 21st Century
Skiing in the 21st Century

12:57
Skiing in the 21st Century

13:00
Skiing in the 21st Century
Skiing in the 21st Century
Skiing in the 21st Century

13:03
Skiing in the 21st Century
Skiing in the 21st Century

You can even warn in advance that you will be late!
How to Feed the Bugs

• Reasoning as in other fields, while ignoring the specificities of Informatics
  – identifying bugs to machine failures
  – computing "bug probabilities"
  – viewing software as "lighter mechanics"
  – adding redundancy by simply doubling the system

• Testing only the basic usage cases
  – bugs most often show in unexpected legal cases
  – untested non-functional bugs make the best security holes

• Forgetting to verify the verification
  – compare to scientific paper reviewing

Bugs are always human failures
The Problem With Informal Specifications

• Contradiction or inconsistency
  – page 12: X is a positive integer variable
  – page 33: if X is negative, ...

• Under-specification
  – forgetting to specify the environment constraints
  – forgetting to specify all the error cases
  – if the programs receives multiple alarms in a very short time, it should react appropriately

• Over-specification
  – giving lots of useless details that make realization harder
    ex. : irrelevant physical characteristics
    unadapted toolset

It is plain impossible to realize a good application when the specifications are imprecise, or even ugly
How to Get Rid of Bugs

• Use appropriate design tools
  – good spec techniques, languages, debuggers, formal tools

• Make everything visible
  – development processes, specs, code, documentation, tests suites, test results, etc

• Independently review everything
  – by certification processes, open source communities, etc

• Perform systematic testing
  – component unit testing, integration testing, regression testing
  – random testing, testing by simulation, testing on real target

21st century: verify formally with automatic and semi-automatic tools
What is Verification?

• Full verification (very ambitious)
  – completely understanding what we want to do
  – completely specifying what the program should do
  – completely understanding what the program does
  – demonstrating perfect match of all these aspects
What is Verification?

• Full verification (very ambitious)
  – completely understanding what we want to do
  – completely specifying what the program should do
  – completely understanding what the program does
  – demonstrating perfect match of all these aspects

• Partial verification (very useful)
  – verify important properties of the system
    there will be no overflow, no out-of-bounds access, etc.
    the elevator will never travel with the door open → safety
    if I call the elevator, it will eventually pick me up → liveness

A difficult goal: compositional verification
Applications and Constraints

flight-control, engines, brakes, fuel, power, etc.
safety-critical $\Rightarrow$ certification

trajectory, attitude, image, telecom
mission-critical $\Rightarrow$ very high quality

telephone, audio, TV, DVD, games
business critical $\Rightarrow$ time to market + quality

pacemakers, insulin control, robot surgeons
life-critical $\Rightarrow$ certification (I hope!)
• C has become THE high-level assembly language
  – many C programs are automatically generated
  – compiler bugs can have very bad consequences

• But C is a complex language
  – arithmetico-logical expressions, type coercions, arbitrary pointers with arithmetic, bit-fields, etc.
  – 191 undefined behaviors, 52 unspecified behaviors, left to implementation choices (!)

• And generated code performance is crucial
  – many optimizations rely on potentially incorrect static analyses
Csmith Principles

• Generating complex but correct C programs
  – random generation from C grammar, test-based filtering
    → delicate aspects: scoping, pointers, operations, const, volatile, etc.
  – limitations: no strings, floating-point, dynamic allocation, function pointers
  – 1 million programs generated!

• Crash test and differential test of compilers
  – direct detection of crashes and internal errors
  – comparing the execution of the generated code by 12 compilers:
    GCC, LLVM, commercial compilers, CompCert (X. Leroy)

Hundreds of bugs found, especially in intermediate phases (transformation and optimization)
Different compilers have different bugs!
Only one survivor: CompCert (X. Leroy, Inria)
Crashes and Assertion Violations

source Xuejun Yang, Yang Chen, Erich Eide et John Reger

Prague, G. Berry, 15/10/2015
Wrong Generated Code

source Xuejun Yang, Yang Chen, Erich Eide et John Reger

CompCert by Xavier Leroy : 0 bug !
Surprizing? No, it is formally certified (in Coq)
Dijkstra's Fundamental Remark

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence

Edsger W. Dijkstra,
The Humble Programmer (1972)
Communications of the ACM 15 (10), 972: pp. 859–866

Can we do better? Yes!
By viewing specifications and programs as mathematical questions to be solved with formal verification software (itself verified)

Beware: this requires really formal mathematics
Leslie Lamport's Mutos (1)

- **Writing** is nature’s way of letting you know how sloppy your thinking is

  *Guindon*

- **Mathematics** is nature’s way of letting you know how sloppy your writing is

  *Leslie Lamport, The TLA+ Book*

- **Formal mathematics** is nature’s way of letting you know how sloppy your mathematics is

  *Leslie Lamport, The TLA+ Book*
Anatomy of a Modern Formal Method

- Algorithmic thinking
- Design and programming
- Reasoning on correctness

- Mathematical and intuitive kernel

- Machines
- Languages
- Implementation
- Optimization
- Verification
In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

\[ u' = u + v \quad s' = s + 1 \]

factorial \hspace{2cm} multiplication
### Assertion Table

<table>
<thead>
<tr>
<th>STORAGE LOCATION</th>
<th>(INITIAL) A $k = 6$</th>
<th>B $k = 5$</th>
<th>C $k = 4$</th>
<th>(STOP) D $k = 0$</th>
<th>E $k = 3$</th>
<th>F $k = 1$</th>
<th>G $k = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td></td>
<td>r</td>
<td>r</td>
<td>s</td>
<td>r</td>
<td>s + 1</td>
<td>s</td>
</tr>
<tr>
<td>28</td>
<td>n</td>
<td></td>
<td></td>
<td>r</td>
<td></td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td></td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td></td>
<td>s</td>
<td>r</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td></td>
<td>l</td>
<td>(s + 1)</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TO B WITH $r' = 1$ $u' = 1$**

**TO C**

**TO D**

IF $r = n$

TO E

IF $r < n$

**TO G**

**TO F**

**TO B WITH $r' = r + 1$**

IF $s \geq r$

TO E

WITH $s' = s + 1$

IF $s < r$

**Terminaison proof:** find a decreasing ordinal
A Hard Termination Proof

\[ f(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\text{if even}(n) \text{ then } f(n/2) & \text{else if even}(n) \\
\text{else } f(3*n+1) & \text{else}
\end{cases} \]

Open problem: terminates for any \( n > 0 \) ?

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
One can completely specify what the program should do

- **Sort**($L$) : formal definition of sorting

  permutation $\pi$ : bijection of $[0..n]$ onto itself
  
  $1 2 3 4 5 \rightarrow 3 5 2 1 4$

  **Definition** : $L'$ is the sort of $L$ of length $n$ iff

  $\forall i < n-1. \ L'[i] \leq L'[i+1]$

  $\exists \pi. \ \forall i \leq n. \ L'[\pi(i)] = L[i]$

- **Fact** ($n$) : the factorial function $n!$
Merge Sort: Proof by Induction

12 17 10 23 33 77 83 11 39 45 14 18 15 31 91 24

12 17 10 23 33 77 83 11 \rightarrow 10 11 12 17 23 33 77 83

39 45 14 18 15 31 91 24 \rightarrow 14 15 18 24 31 39 45 91
Merge Sort: Proof by Induction

Base case: a list of length 1 is sorted
Induction 1: both sublists are sorted
Induction 2: their merge is sorted
A Bad Induction Hypothesis

Theorem: a binary tree with \( n \) vertices has at most \( n \) edges

Base case:
- 1 vertex
- 0 < 1 edges

Induction:
- \( p+q+1 \) vertices
- < \( p+q+2 \) edges

11 vertices
10 edges

Prague, G. Berry, 15/10/2015
A Good Induction Hypothesis

Theorem: a binary tree with \( n \) vertices has exactly \( n-1 \) edges.

**Base Case:**
- 1 vertex
  - 0 = 1 - 1 edges

**Induction:**
- \( p+q+1 \) vertices
  - \((p-1) + (q-1) + 2\)
  - \( = (p+q+1) - 1 \) edges

Prague, G. Berry, 15/10/2015
int fact (int n) {
    int i, r = 1;
    for (i = 2; i <= n; i++) {
        r = r * i;
    }
    return r;
}

How to prove that fact computes the factorial?
Floyd-Hoare Assertion Logic

```c
int fact (int n) {
    { n ≥ 1 } // hypothesis
    int i = 1; r = 1,;
    { r = i! } // loop invariant
    for (i = 2; i <= n; i++) {
        { r = (i-1) ! } r = r*i;
        { r = i! } r = i !
    }
    { r = i!, i = n }
    return r; { fact(n) = n ! }
}
```
The main Formal Methods (1)

- **Static Analysis**
  - *Fancy type-checking*: Caml, Haskell, F#, etc.
  - *Abstract Interpretation* (Cousot):
    - assertions, absence of run-time errors and stack overflow, etc. etc.
    - *industrial*: The Mathworks, AbsInt
      - Ariane, Airbus A380, automotive, ...

- **Automatic model-checking**
  - partial or total exploration of the (huge) state space
    - *explicit / randomizes*: SPIN, CADP (protocols, distributed algorithms)
    - *implicit* (BDDs, SAT, SMT): BLAST, Z3 (Microsoft), ...
    - *temporal*: Uppaal, HyTech
      - *heavy industrial use*
  - *symbolic rewriting*: ProVerif (security protocols), ...

**Advantage**: do not require advanced user expertise
**Temporal Model-Checking**

**Specification**: should leave freedom, since many possible algorithms

**Safety**: the elevator never travels with the door open

\( \square (\text{Moves} \Rightarrow \text{DoorClosed}) \)

- always
- instantaneous predicates

Counter-example generation for false properties
Temporal Model Checking

Liveness: the elevator will eventually reach a floor where it has been called

□ (Call[i] ⇒ ♦ Reach[i])

always \quad eventually

Harder, because talks about an arbitrary future

testable if nb floors known

not if it is a parameter
Automatic Test Generation

Negative question: it is impossible to visit all floors

Negative answer: wrong! Just do the following sequence

Positive result: Thanks!

Many industrial applications: circuit CAD, networks on chips, drivers, protocols, distributed algorithms, critical real-time software, etc.
The Main Formal Methods (2)

• Proof Assistants (HOL, Isabelle, Coq, Rodin, TLA+, ..)
  – implement powerful mathematical logics (classical / intuitionistic)
  – interactive tools to build and verify proofs
  – partial proof automation
  – automatic code extraction (for Coq)

• Success examples
  – industrial: Pentium arithmetic (Harrison), AMD
  – industrial: B, Event B (Abrial): subways (Météor, Lyon), trains, etc.
  – industrial: Sel4 (NICTA), Minix (Prove&Run): OS kernels
  – CompCert (Leroy): Coq-verified compiler

Advantage: huge power
Drawback: requires very specific expertise
seL4 Verified Microkernel Architecture

Haskell prototype → automatic translation → Isabelle

Abstract specification

Executable specification

refinement proof

refinement proof

High-performance C code

10% slower than standard L4
750,000 copies installed (2010)!
11 intermediate languages, 10 proven transformations, use of external algorithms, results proved correct
CompCert Code Extraction

• Coq: an algorithm can only be written together with its termination proof

• The actual CAML executable code of the algorithm is automatically extracted from the proof

Curry-Howard principle: computing $\iff$ proving
Mathematics on Computer

- 1852 Guthrie
- 1976 Appel – Haken
- 2005 Gonthier (en Coq)
Mathematics on Computer

- 1852 Guthrie
- 1976 Appel – Haken
- 2005 Gonthier (en Coq)
Mathematics on Computer

- 1852 Guthrie
- 1976 Appel – Haken
- 2005 Gonthier (en Coq)

Proof omitted, long but classical....

2013 : Feit-Thompson’s Odd Order Theorem
255 pages of heavy mathematics → Coq!
Objectifs

• Complete or replace testing by mathematical proofs, with at least the degree of confidence of human mathematical proofs

• Better (and doable): Assist / automatize proofs

• Generate counter-examples for false properties

• Make formal verification accessible to engineers and scientists

• Integrate formal verification in usual development flows

• Understand when formal verification is practically justified

And use formal verification as soon as possible and as long as possible