Using theorem proving in industry

John Harrison

Intel Corporation

- The cost of bugs
- Formal verification
- Machine-checked proof
- Automatic and interactive approaches
- HOL Light
- Floating point verification
- Theorem prover features
- Conclusions

The human cost of bugs

Computers are often used in safety-critical systems where a failure could cause loss of life.

- Heart pacemakers
- Aircraft
- Nuclear reactor controllers
- Car engine management systems
- Radiation therapy machines
- Telephone exchanges (!)
- ...

Financial cost of bugs

Even when not a matter of life and death, bugs can be financially serious if a faulty product has to be recalled or replaced.

- 1994 FDIV bug in the Intel®Pentium® processor: US $500 million.
- Today, new products are ramped much faster...

So Intel is especially interested in all techniques to reduce errors.

Complexity of designs

At the same time, market pressures are leading to more and more complex designs where bugs are more likely.

- A 4-fold increase in bugs in Intel processor designs per generation.
- Approximately 8000 bugs introduced during Pentium 4 design process.

Fortunately, pre-silicon detection rates are now at least 99.7%.

Just enough to tread water...

Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

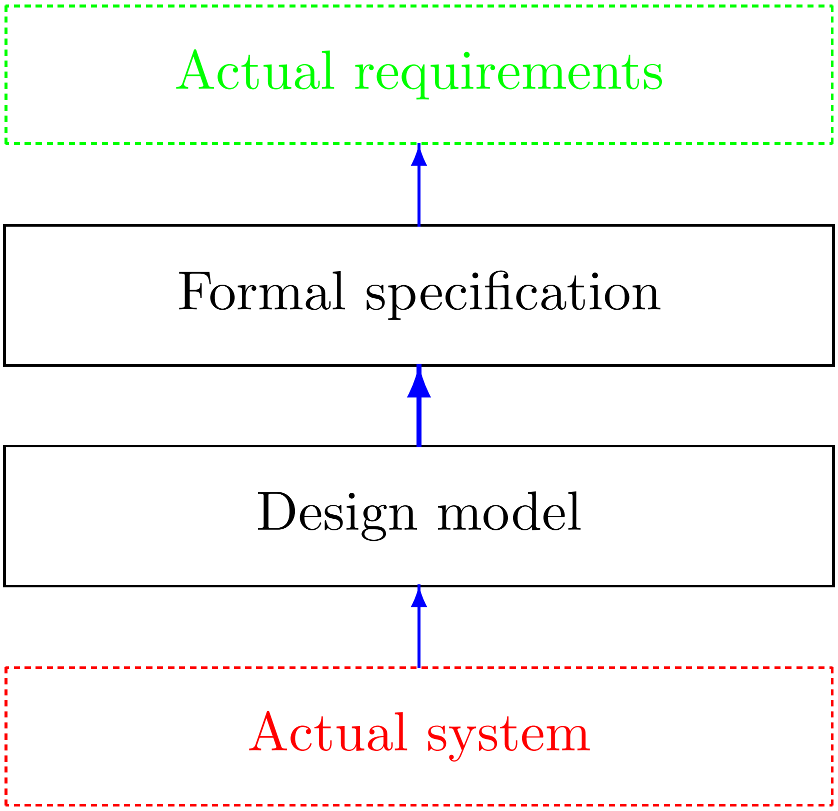
- Slow — especially pre-silicon
- Too many possibilities to test them all

For example:

- 2^{160} possible pairs of floating point numbers (possible inputs to an adder).
- Vastly higher number of possible states of a complex microarchitecture.

Formal verification

Formal verification: mathematically prove the correctness of a *design* with respect to a mathematical *formal specification*.



Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

- Difficult and time-consuming.
- Only as reliable as the formal models used.
- How can we be sure the proof is right?

Analogy with mathematics

Sometimes even a huge weight of empirical evidence can be misleading.

- $\pi(n)$ = number of primes $\leq n$
- $li(n) = \int_0^n du/ln(u)$

Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often.

No change of sign at all had ever been found despite testing up to $n = 10^{10}$ (in the days before computers).

Similarly, extensive testing of hardware or software may still miss errors that would be revealed by a formal proof.

Formal verification is hard

Writing out a completely formal proof of correctness for real-world hardware and software is difficult.

- Must specify intended behaviour formally
- Need to make many hidden assumptions explicit
- Requires long detailed proofs, difficult to review

The state of the art is quite limited.

Software verification has been around since the 60s, but there have been few major successes.

Faulty hand proofs

"Synchronizing clocks in the presence of faults" (Lamport & Melliar-Smith, JACM 1985)

This introduced the Interactive Convergence Algorithm for clock synchronization, and presented a 'proof' of it.

- Presented five supporting lemmas and one main correctness theorem.
- Lemmas 1, 2, and 3 were all false.
- The proof of the main induction in the final theorem was wrong.
- The main result, however, was correct!

Machine-checked proof

A more promising approach is to have the proof checked (or even generated) by a computer program.

- It can reduce the risk of mistakes.
- The computer can automate some parts of the proofs.

There are limits on the power of automation, so detailed human guidance is usually necessary.

The spectrum of theorem provers

From interactive proof checkers to fully automatic theorem provers.

AUTOMATH (de Bruijn)

Stanford LCF (Milner)

Mizar (Trybulec)

. . .

. . .

PVS (Owre, Rushby, Shankar)

. . .

. . .

ACL2 (Boyer, Kaufmann, Moore)

Otter (McCune)

Automation vs. expressiveness

Tools like Boolean tautology checkers and symbolic model checkers are:

- Completely automatic
- Efficient enough for nontrivial problems
- Incapable even of expressing, let alone proving, many interesting properties.

On the other hand, proof checkers like Mizar:

- Can prove essentially any mathematical theorem in principle
- Require detailed and explicit human guidance even for relatively simple problems.

To verify interesting floating-point algorithms, we need automation *and* expressiveness.

HOL Light

HOL Light is based on the approach to theorem proving pioneered in Edinburgh LCF in the 70s.

- All theorems created by low-level primitive rules.
- Guaranteed by using an abstract type of theorems; no need to store proofs.
- ML available for implementing derived rules by arbitrary programming.

The system can be extended reliably without making unsafe modifications

The user controls the means of production (of theorems).

Other LCF theorem provers

There are many versions of HOL:

- HOL88
- hol90
- ProofPower
- HOL Light
- hol98

and several other provers based on LCF:

- Coq
- Isabelle
- Nuprl

HOL Light primitive rules (1)

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\, s) = (\lambda x.\, t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\, t)x = t}\ \texttt{BETA}$$

HOL Light primitive rules (2)

$$\frac{}{\{p\} \vdash p} \quad \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \quad \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q} \quad \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]} \quad \texttt{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]} \quad \texttt{INST\_TYPE}$$

Floating point verification

We've used HOL Light to verify the accuracy of floating point algorithms (used in hardware and software) for:

- Division and square root
- Transcendental function such as *sin*, *exp*, *atan*.

This involves background work in formalizing:

- Real analysis
- Basic floating point arithmetic

We'll give some examples to show the importance of some of HOL Light's features.

Need for pre-proved mathematics

Many floating-point algorithms are based on particular formulas for transcendental functions. For example, to calculate the tangent of a number close to $\pi/2$, we use the cotangent expansion, valid for $0 < |x| < \pi$:

$$cot(x) = 1/x - \frac{1}{3}x - \frac{1}{45}x^3 - \frac{2}{945}x^5 - \ldots$$

To verify the error when approximating $tan(\pi/2 + x)$ with some truncation of this series requires quite a lot of real analysis, e.g. differentiable functions, continuity, Taylor series, general theorems on reversing orders of summations...

HOL's pre-proved real analysis

- Definitional construction of real numbers
- Basic topology
- General limit operations
- Sequences and series
- Limits of real functions
- Differentiation
- Power series and Taylor expansions
- Transcendental functions
- Gauge integration

Examples of useful theorems

```
|- sin(x + y) =
   sin(x) * cos(y) + cos(x) * sin(y)

|- tan(&n * pi) = &0

|- &0 < x /\ &0 < y
   ==> (ln(x / y) = ln(x) - ln(y))

|- f contl x /\ g contl (f x)
   ==> (g o f) contl x

|- (!x. a <= x /\ x <= b
        ==> (f diffl (f' x)) x) /\
   f(a) <= K /\ f(b) <= K /\
   (!x. a <= x /\ x <= b /\ (f'(x) = &0)
        ==> f(x) <= K)
   ==> !x. a <= x /\ x <= b ==> f(x) <= K
```

The need for automation

Many industrial verification proofs are enormously messy and complicated, involving hundreds of millions of logical inferences.

But many of them are for quite routine tasks for which a decision method is available, e.g. linear real arithmetic:

$$\begin{aligned}
&a \leq x \land b \leq y \land \\
&|x - y| < |x - a| \land \\
&|x - y| < |x - b| \land \\
&(b \leq x \Rightarrow |x - a| \leq |x - b|) \land \\
&(a \leq y \Rightarrow |y - b| \leq |y - a|) \\
&\Rightarrow (a = b)
\end{aligned}$$

It's also useful to be able to perform traditional first order logical automation, to avoid a lot of tedious application of inference rules collecting lemmas together.

HOL's automation

- Simplifier for (conditional, contextual) rewriting.
- Tautology checker.
- Automated theorem provers for pure logic, based on tableaux and model elimination.
- Tools for definition of (infinitary, mutually) inductive relations.
- Tools for definition of (mutually) recursive datatypes
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.
- Nonlinear polynomial quantifier elimination over $\mathbb{C}$

Automation examples

Linear arithmetic:

```
REAL_ARITH
  ‘a <= x /\ b <= y /\
   abs(x - y) < abs(x - a) /\
   abs(x - y) < abs(x - b) /\
   (b <= x ==> abs(x - a) <= abs(x - b)) /\
   (a <= y ==> abs(y - b) <= abs(y - a))
   ==> (a = b)‘;;
```

First order logic (realistic examples are too big...)

```
prove
 (‘(!x y z. P x y /\ P y z ==> P x z) /\
   (!x y z. Q x y /\ Q y z ==> Q x z) /\
   (!x y. Q x y ==> Q y x) /\
   (!x y. P x y \/ Q x y)
   ==> (!x y. P x y) \/ (!x y. Q x y)‘,
  MESON_TAC[]);;
```

The need for programmability

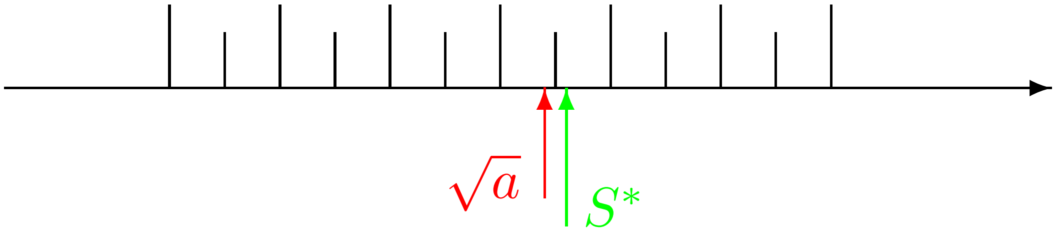
Many of the inference patterns that arise in floating-point work don't correspond to standard decidable problems, but there are special algorithms to solve them. For example:

- Bounding successive rounding errors using relative error analysis and triangle inequality.
- Computing the approximation error in approximating a mathematical function by a polynomial with floating-point coefficients.
- Solving diophantine equations defining difficult cases and proving by exhaustive case analysis that an algorithm is always correct.

These are examples we've automated to the point where they are 'push-button'. Under the surface, they may involve millions of inferences...

Square root perfect rounding

Several square root algorithms work by a final rounding of a more accurate intermediate result S^*. For perfect rounding, we should ensure that the two real numbers $\sqrt{a}$ and S^* never fall on opposite sides of a midpoint between two floating point numbers, as here:



Rather than analyzing the rounding of the final approximation explicitly, we can just appeal to general properties of the square root function.

Exclusion zones

It would suffice if we knew for any midpoint m that:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* cannot lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m IN midpoints fmt
          ⇒ abs(x - y) < abs(x - m))
   ⇒ (round fmt Nearest x =
          round fmt Nearest y)
```

And this is possible to prove, because in fact every midpoint m is surrounded by an ‘exclusion zone’ of width $\delta_m > 0$ within which the square root of a floating point number cannot occur.

However, it turns out that out that there are still some cases where the error in some of our algorithms might be too large.

Difficult cases

However, one can show that all the difficult cases have mantissas m that are solutions of simple diophantine equations, typically of the form:

$$2^p m = k^2 + d$$

It's not difficult to program HOL to enumerate all solutions to such equations for particular p and d, and then to exhaustively test the algorithm for correctness on these numbers (typically a few hundred).

Thus, we have a general mathematical proof covering most cases, and use number theory to isolate the possible exceptions and check them specially — all automatically, and all within a strict formal proof.

Conclusions

- Formal verification of mathematical software is industrially important, and can be attacked with current theorem proving technology.
- A large part of the work involves building up general theories about both pure mathematics and special properties of floating point numbers.
- It is easy to underestimate the amount of pure mathematics needed for obtaining very practical results.
- The mathematics required is often the sort that is not found in current textbooks: very concrete results but with a proof!
- Using HOL Light, we can confidently integrate all the different aspects of the proof, using programmability to automate tedious parts.