Formal Verification In Industry (I)

John Harrison

Intel Corporation

- Formal verification
- Importance of hardware verification
- Approaches to hardware verification
- Combinational comparison
- BDDs
- Symbolic simulation and STE
- Temporal logic model checking
- Theorem proving
- The best of both worlds?

Formal Verification

Traditionally, errors in hardware and software have been discovered empirically, by testing them in many possible situations.

However, the number of possible situations is usually so large that we can only exercise a tiny proportion of them.

For example, there are about 2^{80} double extended precision floating point numbers. Testing an operation on all of them will probably never be feasible, even if it's only unary.

Pre-silicon testing of microprocessor designs is especially limited, since everything is run on simulators orders of magnitude slower than real hardware.

Formal verification is an alternative that involves trying to *prove* mathematically that a computer system will function as intended.

Exhaustiveness

In mathematics, a general proposition can't be *proved* by testing many possible cases. A rigorous proof is something different.

Sometimes even a huge weight of numerical evidence can be misleading. For example, Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often, where $\pi(n)$ is the number of primes $\leq n$ and

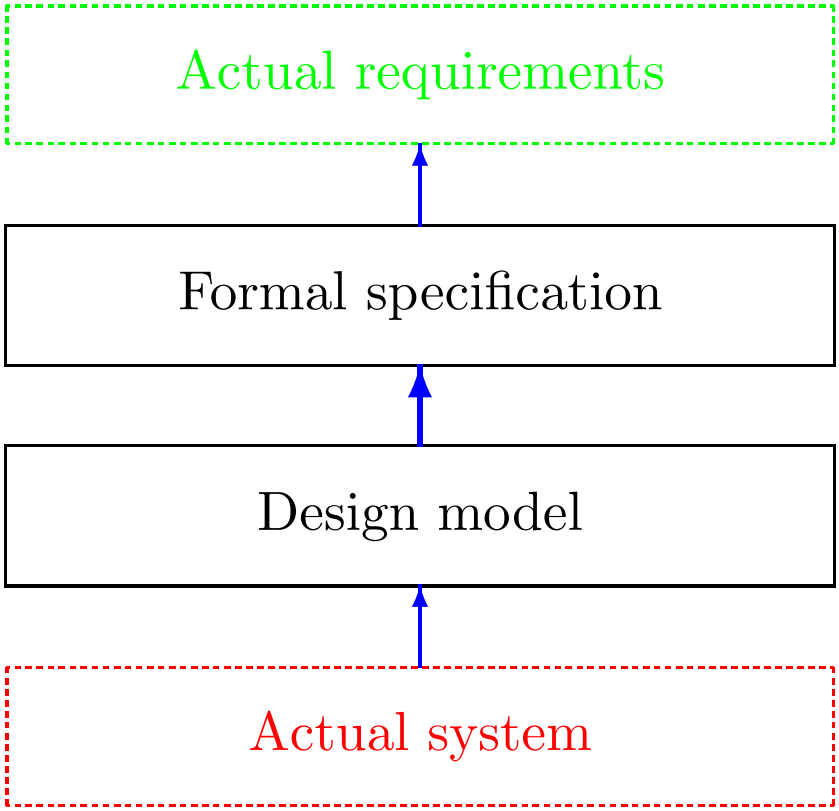
$$li(n) = \int_0^n du/ln(u)$$

This came as a surprise since not a single sign change had been found despite extensive testing of values up to 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 models

Formal verification aims to prove the correctness of a *design* with respect to a mathematical *formal specification*. This still leaves two gaps:



Note that the same criticisms can be levelled at certain kinds of testing. A simulator is not the same as a real chip. Checking against a 'reference implementation' doesn't prove that the reference is correct.

Formal verification is hard

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

One needs to make explicit lots of assumptions and special cases that one often forgets about informally. Moreover, one has to avoid making any mistakes or oversights. This is a major undertaking, even for a small system.

It's not easy to get such long and detailed proofs right, nor for others to read them and be assured of their correctness.

The state of the art, at least in the software world, is quite limited. Software verification has been around since the 60s, but there have been few major successes.

Hobbes quotation

> And as in Arithmetique, unpractised men must, and Professors themselves may often erre, and cast up false; so also in any other subject of Reasoning, the ablest, most attentive, and most practised men, may deceive themselves and inferre false Conclusions; Not but that Reason it selfe is always Right Reason, as well as Arithmetique is a certain and infallible Art: But no one mans Reason, nor the Reason of any one number of men, makes the certaintie; no more than an account is therefore well cast up, because a great many men have unanimously approved it.

From Hobbes's *Leviathan*, 1651.

Computer theorem provers

A more promising approach is to have the proof checked (or even generated) by a computer program. This offers two potential advantages over doing proofs by hand:

- It can reduce the risk of mistakes. The computer can check that the user only proves results in ways known to be sound.
- The computer can make (some parts of) the proof easier than they would be by hand, even automating large parts of it.

In the hardware world the latter has proven to be especially important, and has led to a recent upsurge of interest in formal hardware verification.

Faulty hand proofs

The paper "Synchronizing clocks in the presence of faults" (Lamport & Melliar-Smith, JACM 1985) introduced the Interactive Convergence Algorithm for clock synchronization, and presented a 'proof' of it.

A later attempt to reproduce this by Rushby and von Henke in a mechanical theorem prover (EHDM) discovered serious flaws.

The paper presented five supporting lemmas and one main correctness theorem.

Lemmas 1, 2, and 3 were all false. Lemma 4 was false too, but only because of a minor typographical error. The proof of the main induction in the final theorem was wrong. The main result, however, was correct!

Hardware verification

In recent years, formal hardware verification has become much more important, and used by most or all leading hardware companies, while software verification is still languishing. Why?

First, there seems a greater need for it. We have already pointed out the greater commercial significance of hardware errors. And the complexity of hardware designs is increasing rapidly, as designers rely on techniques such as deep pipelining, out-of-order execution and speculation. Because of the potential for subtle interactions between components, it is increasingly difficult to exercise a realistic set of possibilities by simulation.

Secondly, important aspects of hardware design are amenable to *automated* proof methods, making formal verification easier to introduce and more productive.

Combinational comparison

One important question in hardware design is whether two different *combinational* circuits have the same behaviour (subject perhaps to some conditions on the inputs).

For example, a synthesis tool may produce a circuit automatically. However, typically the designer would like to modify it, e.g. to reduce the gate count or improve timing.

The task of showing that the unoptimized and optimized circuits have the same functional behaviour reduces simply to verifying a formula in *propositional* (Boolean) logic ('tautology checking').

This can be done completely automatically, e.g. by truth tables.

Efficiency

Even though purely propositional theorem proving can 'in principle' be done completely automatically, in practice it can be prohibitively expensive when the formula is large and/or involves many variables.

In fact, the tautology checking of propositional formulas is a (co-)NP complete task, so it seems likely that no algorithm will have good worst-case behaviour.

One way of avoiding this difficulty in combinational comparison is to partition the circuits into many parts and verify corresponding components.

Even so, one would like the underlying propositional reasoning system to be as efficient as possible. Truth tables rapidly become too inefficient in practice.

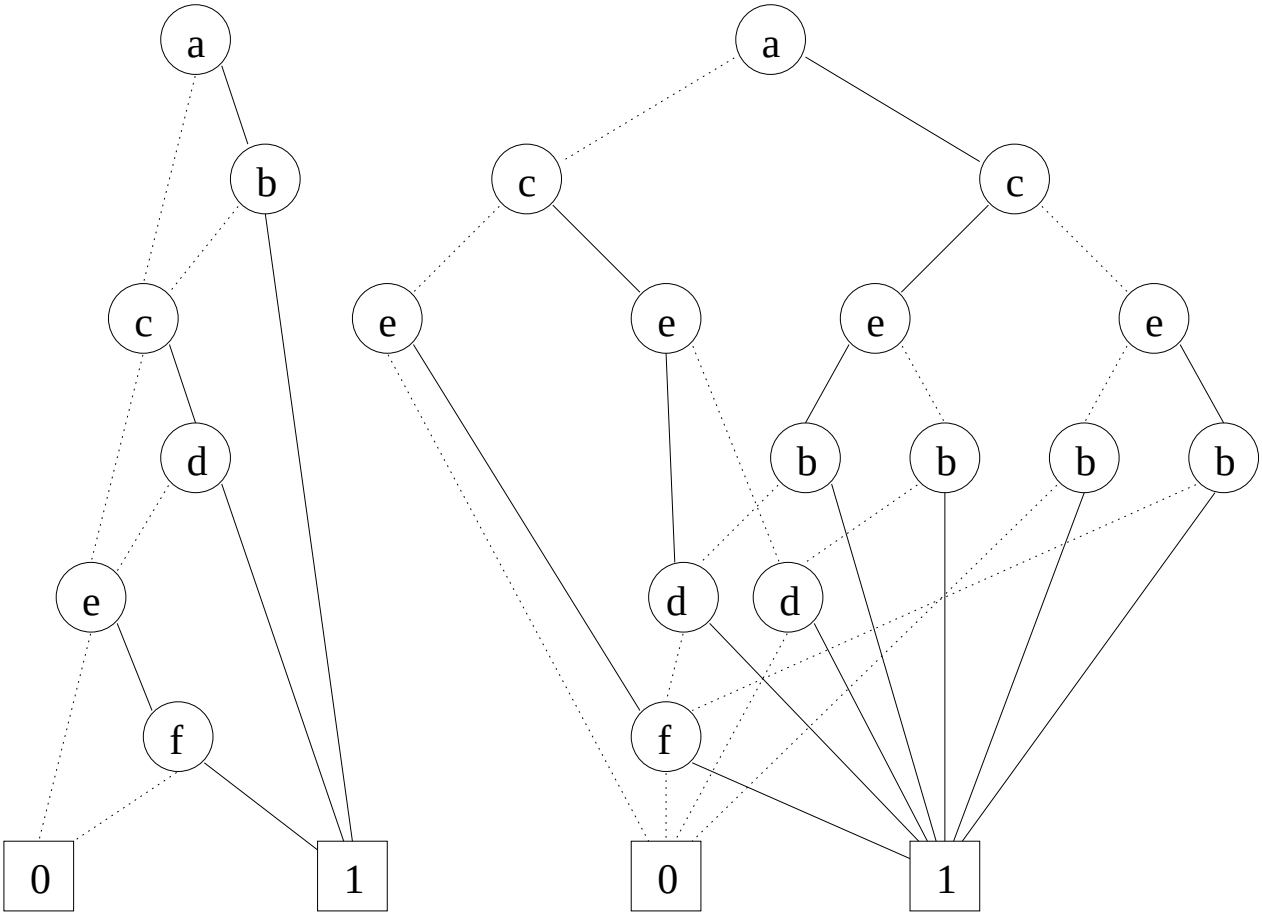
Efficient tautology checking

There are several ways of doing tautology checking that often work out quite well in practice, e.g.

- Classic algorithms like the Davis-Putnam procedure.
- Reduction to an integer programming problem.
- Stålmarck's algorithm
- Binary decision diagrams (BDDs).

In the hardware industry, BDDs are currently the most popular. They have a few other nice features such as giving a *canonical* representation of a formula.

BDDs



Boolean functions are represented by directed acyclic graphs with maximal sharing and a canonical variables ordering.

The variable ordering can make a big difference!

Symbolic simulation

Symbolic simulation is a generalization of traditional logic simulation, where the 'test vectors' are not just constants, but expressions involving variables. This allows a single simulation run to consider broad classes of inputs.

This approach to verification was developed in the 1970s, but the methods for handling the symbolic expressions were primitive and inefficient, and the method did not have much impact.

The introduction of BDDs to represent the expressions made a dramatic difference; they give a canonical representation of arbitrary Boolean expressions, and turn out to be quite compact in many practical situations.

Using symbolic simulation with BDDs, it is possible to perform verification against a specification simply by comparing the results with the expectation, both symbolic expressions.

Symbolic trajectory evaluation

Symbolic trajectory evaluation (STE) is a further development of symbolic simulation.

The user can write specifications in a restricted *temporal logic*, specifying the behaviour over bounded-length *trajectories* (sequences of circuit states).

A typical specification would be: if the current state satisfies a property P, then after n time steps, the state will satisfy the property Q.

The circuit can then be checked against this specification by symbolic simulation.

Temporal logic model checking

In temporal logic model checking, specifications can be written in a more general temporal logic without the limitation of bounded trajectories or explicit time.

The hardware itself is regarded simply as a state transition system. Each state gives rise to a valuation on atomic formulas, saying whether that atomic formula 'holds in a state'.

In CTL (Computation Tree Logic), behaviour can be specified by quantifying both over future time and over the range of possible state transition sequences. For example AGf means 'for all possible paths, f holds in every state'.

To make the model-checking algorithm practical, the state transition system is coded up with combinations of Boolean variables, so that everything can be represented by BDDs.

Limitations of automatic approaches

All the methods we have considered are largely automatic. Moreover, in cases where the verification fails, they can provide explicit counterexamples. However they have important limitations.

First, despite clever tricks like BDDs, one inevitably hits feasibility problems for large circuits, especially those involving wide datapaths. The problem is particularly acute in temporal logic model checking, because one needs a more or less explicit representation of all the states of the system.

Second, even CTL is quite a limited logic, and cannot express many important properties. In particular, none of the methods considered can deal with numbers. This means that arithmetic hardware is merely being verified against a bit-level specification, leaving room for doubt over

whether this correctly specifies arithmetic.

For even higher-level specification, e.g. as in floating point operations, the situation is still worse.

Theorem proving

One can meet these objections by using a general theorem prover.

This can deal with all the high-level mathematics required, and the specification can therefore be written in a more natural way. In fact, the verification can be modularized and structured into layers, with increasingly general levels of specification.

Verification can also be performed generically, e.g. proving n-bit adders correct for arbitrary n rather than some particular value.

However, there is the serious disadvantage that once one goes much further than CTL, validity is no longer decidable in theory and certainly not feasible in practice. Instead, proving theorems needs expert interaction. Moreover, one no longer gets the automatic counterexamples when proofs fail.

The best of both worlds?

Perhaps the ideal is to combine theorem proving and model checking together in a single system:

- The model checker to perform low-level parts of the proof, generate feedback in the form of counterexamples, and raise the level of automation.
- The theorem prover to link to higher-level specifications, perform inductive or generic reasoning and verify the underlying mathematics

Intel has been successfully using a combination of theorem proving and STE for a number of years. Recently, this has been increasingly tightly coupled to achieve a productive environment. The combined system has been applied to verifying the basic floating point operations in the P6.

It is likely that this kind of research will be increasingly important in the future.