Practical Formal Verification at an Industrial Scale

Tom Melham
University of Oxford
Computing Laboratory

Joint Work With
Strategic CAD Labs
Intel Corporation

Pentium 4 Microprocessor

• 55M transistors
• 146mm²
• 0.13µm process
• 3.2GHz

Among the most complex artefacts ever produced by humans...
Moore’s Law

The number of transistors doubles around every 18 months

Design Productivity Crisis

How many people will it take in 2010?
The Verification Crisis

• Computer simulation

• Typical effort

• The verification crisis

Formal Verification

• Use mechanized logic to prove correctness
  • no simulation test cases needed
  • complete coverage, effectively exhaustive simulation

• But it’s not easy
  • scalability, ease of use, getting specifications

• Different philosophies
  • verification (insurance)
  • bug finding (productivity)
  • verification through design
Verification in Practice

• Industrial experience shows
  • theory by itself is not usable (in practice)
  • theory with a system is not usable (in practice)
  • something else is needed - an FV methodology

• A systematic, pragmatic approach to organizing large-scale verification efforts:
  • clearly stated plan for the sequence and purpose of the many interdependent activities involved
  • guiding structure for the verification code artifacts to be produced

Methodological Principles

• Realistic
  • complete specifications are usually not available
  • access to design engineers is always limited

• Transparent and Sound
  • should be clear what has and has not been proved
  • must be sound, no false positives

• Structured
  • helps new users learn
  • increases productivity of experienced users

• Incremental
  • must be able to measure progress
  • effort should develop ‘debugging value’ early
  • past work should help with recovery when proofs break
Methodological Principles

• Provides good feedback
  • bulk of verification effort is debugging
  • optimize for proof failure, not success
  • provide focused feedback and tight debugging loop

• Top-down and bottom-up
  • top-down for problem reduction/abstraction
  • bottom-up for understanding design and tool capacity limits

• Supports regression
  • verification artifacts should be maintainable
  • and easily adaptable to track design or specification changes

• Allows effort reuse
  • verification is human-intensive
  • amortize cost over changes or multiple designs
  • proof effort should be relatively circuit independent

Forte Formal Verification System

Intel’s interactive verification environment

- Functional Programming (FL/refLect)
- Higher Order Logic
- Theorem Proving
- STE
- GSTE
- SAT
- BDDs
- Symbolic Simulation

Forte ≈
- temporal logic
- symbolic simulation + abstraction
- lightweight theorem proving
- functional scripting
- reflection
Some Forte Verifications

- Verification of gate-level floating point implementations against IEEE specification: `FADD`, `FSUB`, `FMUL`, `FDIV`, `FSQRT`, ...
- IA32 instruction-length decoder.

- Pentium 4 FV effort found several ‘high quality’ bugs
  - FP multiply data space bug
  - interaction between FP ops in different threads, corrupting data
  - FP overflow/underflow flags incorrect in certain processor modes
  - overall FV effort (not just FPU, not just Forte) found ~20 high quality bugs hard to detect by dynamic testing [Bentley, DAC’01].

- Intel has used these techniques on two generations of lead IA32 processors and their proliferations.

Role of Functional Programming

- Specification
  - data modeling (e.g. floating point operations)
  - stipulating I/O behavior
  - stipulating timing behavior

- Scripting
  - encapsulating circuit details
  - generating test cases and running simulations
  - invoking and controlling model checkers
  - scripting theorem proving strategies
  - programming exploration of counterexamples

- Tool Building
  - prototyping model checking strategies
  - making new deductive theorem proving procedures
Evaluation

• Scalar data

  : \texttt{defix} '+';
  \quad : :: \texttt{bool list} \to \texttt{bool list} \to \texttt{bool list}

  : \texttt{let} \ a = [F,T,T,T]; \quad // \ 0111
  \quad a :: \texttt{bool list}

  : \texttt{let} \ b = [F,F,F,T]; \quad // \ 0001
  \quad b :: \texttt{bool list}

  : a '+' b;
  \quad \ [T,F,F,F] :: \texttt{bool list}

Symbolic Evaluation

• Symbolic data

  : \texttt{let} \ A = \texttt{map variable} \ ["a3","a2","a1","a0"];
  \quad A :: \texttt{bool list}

  : \texttt{let} \ B = \texttt{map variable} \ ["b3","b2","b1","b0"];
  \quad B :: \texttt{bool list}

  : A '+' B;
  \quad \ [b2\&b3\&a2\&a3 + b1\&b3\&a1\&a2\&a3 + b0\&b3\&a0\&a1\&a2\&a3 + !b0\&!b1\&!b2\&!b3\&a3 \ OR \ ... ,
  \quad b1\&b2\&a1\&a2 + b0\&b2\&a0\&a1\&a2 + !b0\&!b1\&!b2\&a2 +
  \quad !b1\&!b2\&a0\&a2 \ OR \ ... ,b0\&b1\&a0\&a1 +
  \quad !b0\&!b1\&a1 + !b1\&!a0\&a1 + !b0\&!b1\&!a1 +
  \quad b0\&!b1\&a0\&!a1 \ OR \ ... ,!b0\&a0 + b0\&!a0] :: \texttt{bool list}

  :
Reflection - reFLect

- Reflection
  - programs are data, in the same language
  - programs can construct and analyze programs
  - programs can run the constructed programs

- Like LISP/ACL2 quotation, but with types:

\[
\langle 1 + 2 \rangle : \text{term} \quad \text{- an abstract syntax tree:}
\]

\[
\langle 1 + ^{(2)} \rangle = \langle 1 + 2 \rangle \quad \text{- term splicing}
\]

\[
\text{let swap } \langle ^{x} + ^{y} \rangle = \langle ^{y} + ^{x} \rangle \quad \text{- pattern matching}
\]

Why Reflection?

- For our Forte applications, we want to
  - reason about arbitrary FL programs
  - reason about FL specifications of hardware
  - analyze & transform hardware models written in FL
  - intimately combine theorem proving and program execution
  - intimately combine model checking and theorem proving

- All require programs to access syntax of FL programs.

- Exploit reflection in theorem proving
  - term language of logic = reFLect = theorem prover metalanguage
  - evaluation in reFLect ⇒ provable equation in theorem prover
Motivating Example - FP Adder

• FPU from Intel Pentium Pro processor
  • large block of RTL from architects and designers
  • highly optimized for speed
  • to be verified ‘as is’, with no modification of design

• Functionality to verify
  • IEEE-compliant FP addition and subtraction
  • multiple precision: single, double, extended
  • all rounding modes: towards 0, +∞, -∞, to nearest

Proof Strategy Overview

• Create a reference model
  • must be formal
  • must be executable

• Show that the model
  • satisfies the IEEE spec
    - theorem proving
  • is equivalent to the RTL
    - STE model checking
Example – Rounding Specification

```ocaml
let RND pc rc s sgf =
  // Extract lsb, guard, round sticky bits.
  let L = Lsb pc sgf in
  let G = Guard pc sgf in
  let RS = RoundS pc sgf in
  // Conditionally add one to LSB
  let rbit =
    (rc '=' TO_ZERO)    => F
  | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
  | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
  | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
  | F in
  // Result truncates mantissa to precision specified
  // by pc, adds rbit and pads result with zeros.
  Result rbit pc sgf;
```

Circuit API

- The ‘glue’ between reference model and the RTL
- Encapsulates signal names, interface, timing, protocol, ...

- Challenges:
  - complex interface
  - complex internal behavior
  - protocols poorly documented
  - protocols and circuit will change
Verifying RTL

- Symbolic trajectory evaluation
- Based on symbolic simulation
  - smooth transition between simulation and verification
- Abstraction through X values

![Diagram showing flow from stimulus to response and all counterexamples]

Technical Challenges

- Key technical insights
  - Split input data space to avoid data-dependent shifts
    - Use parametric representation
  - Overlay STE verifications with different BDD orderings
    - one for LZA circuit, one for result
  - Certain other technical devices...

- Methodology & tool do not replace scientific insight, technical skill, and innovation!
Data-Dependent Shifts

Adder BDDs blow up because of variable shift

Solution - Input Case Splitting

\[ e_2 - e_1 = 0 \]
\[ e_2 - e_1 = 1 \]
\[ e_2 - e_1 = 2 \]
\[ \ldots \]
\[ e_2 - e_1 = 64 \]
\[ e_2 - e_1 > 64 \]
Combining Cases - Theorem Proving

Goaled Theorem Prover

|- ∃h. STE ckt h A C

STE inference rules

|- ∃h. STE ckt h A B
|- ∃h. STE ckt h B C

logic

|- STE ckt opt1 A B
|- STE ckt opt2 B C

reFLect Interpreter

n = p

n → p

STE ckt opt1 A B → True

STE ckt opt2 B C → True

Extracting Logical Content

• Parametric
  • represent predicate
    \[ \begin{array}{c}
P \subseteq B \times B \times \ldots \times B \\
\end{array} \]
  by function with image \( P \).

• Example
  \[ \begin{array}{c}
  a \ b \ c \ d \\
  \{ (1, 0, 0, 1), \\
  (1, 0, 0, 0), \\
  (0, 1, 0, 1) \} \\
  \end{array} \]

\[ p, \neg p, \ 0, \neg p \lor q \]

• Parametric verification
  • want to check
    \[ \begin{array}{c}
P[xs] \supset STE A[\{xs\}] C[\{xs\}] \\
\end{array} \]

  • parametric representation:
    \[ fs[vs] = param(xs, P[xs]) \]

  • more efficient verification:
    \[ STE A[fs[vs]] C[fs[vs]] \]
### Input Constraints

\[
\text{let thm} = \\
\quad \text{let } fs = \text{param}(xs, P) \text{ in} \\
\quad \quad \text{STE ckt opt } A[fs/xs] B[fs/xs];
\]

\[
\vdash P \supset \exists h. \text{STE ckt } h \ A \ B
\]

\[
\text{def} \quad \vdash \text{let } fs = \text{param}(xs, P) \text{ in} \\
\text{STE ckt opt } A[fs/xs] B[fs/xs]
\]

\[
\vdash \text{thm}
\]

\[
\text{thm } \rightarrow \text{True}
\]

### Input Case Splitting

\[
\text{let thm } i = \\
\quad \text{let } fs = \text{param}(xs, P \ i) \text{ in} \\
\quad \quad \text{STE ckt opt } A[fs/xs] B[fs/xs];
\]

\[
\vdash \exists h. \text{STE ckt } h \ A \ B
\]

\[
\vdash P \ 1 \lor P \ 2 \lor P \ 3
\]

\[
\vdash P \ 1 \supset \exists h. \text{STE ckt } h \ A \ B
\]

\[
\vdash P \ 2 \supset \exists h. \text{STE ckt } h \ A \ B
\]

\[
\vdash P \ 3 \supset \exists h. \text{STE ckt } h \ A \ B
\]

\[
\text{thm } 1 \rightarrow \text{True} \quad \text{thm } 2 \rightarrow \text{True} \quad \text{thm } 3 \rightarrow \text{True}
\]
The Completed RTL Verification

- Final proof
  - 342 cases, verified in parallel on workstation network
  - scripted in FL

- Reuse on subsequent designs
  - reference model
  - case splitting strategy
  - circuit API

- Now a routine technique, usable by non-experts

Some Forte Conclusions

- Industrial-scale formal hardware verification
  - an interactive, *programming*, activity
  - deep insight into tool capabilities *and* the design
  - essential to separate essence from circuit details
    - human comprehension of and reasoning about essence
    - contain and automate away the accidental details

- Explicit methodology principles, realistic

- Effective tools
  - open framework, heavily customised by scripting
  - specifically supports the methodology/approach
‘Formal Methods’ - on Hard Problems

• Not automatic, not manual
• Human
  See and express the essence of the design analysis in domain-friendly terms

• Machine
  Execute/enact the analysis, ‘automating away’ formal & implementation details

• Verification tool support
  • express the needed abstractions
  • fluently explore verification strategies
  • tie abstractions+strategies to the implementation

  • open, programmable, tool
  • white-box integration of components in a scripting environment

Thank You