Why formal verification remains on the fringes of commercial development

Arvind
Computer Science & Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Formal Methods, Turku, Finland
May 28, 2008

With help from Nirav Dave & Mike Katelman
A designer’s perspective

- The goal is to design systems that meet some criteria such as cost, performance, power, compatibility, robustness, ...
- The design effort and the time-to-market matter ($$$)

Can formal methods help?
My Viewpoint

- The degree of correctness required depends upon the application
  - Different applications require vastly different formal techniques
  - Formal tools must be tied directly to high-level design languages
  - Formal techniques should be presented as debugging aids during the design process
  - A designer is unlikely to do anything for the sake of helping the post design verification

The real success of a formal technique is when it is used ubiquitously without the designer being aware of it, e.g., type systems
This talk will show via examples

- How the correctness requirements differ vastly for different applications

- How Bluespec helps the designers in each case
What is Bluespec?
Bluespec: State and Rules organized into *modules*

All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit. *Behavior* is expressed in terms of guarded atomic actions on the state:

Rule: guard → action

Rules can manipulate state in other modules only *via* their interfaces.
GAA Execution model

Repeatedly:
- Select a rule to execute
- Compute the state updates
- Make the state updates

Hardware Implementation: We need to restrict this behavior to be deterministic

To get “good” hardware, we must schedule multiple rules in a cycle concurrently, while preserving the one-rule-at-a-time semantics

Unity [Chandy & Misra]
Compiler model:
Every rule executes in one clock cycle

- Modules (Current state)
- Rules
  - $\delta_1$
  - $\pi_1$
  - $\delta_n$
  - $\pi_n$

- Scheduler
  - "CAN_FIRE"
  - "WILL_FIRE"

- Modules (Next state)

Compiler generates a scheduler to pick a non-conflicting subset of "ready" rules.
Bluespec: Two-Level Compilation

Bluespec (Objects, Types, Higher-order functions)

Level 1 compilation

Rules and Actions (Term Rewriting System)

Level 2 synthesis

Object code (Verilog/C)

Lennart Augustsson
@Sandburst 2000-2002

• Type checking
• Massive partial evaluation and static elaboration

Now we call this Guarded Atomic Actions

James Hoe & Arvind
@MIT 1997-2000

• Rule conflict analysis
• Rule scheduling
Example 1:

IP Lookup in a router

Simple, deterministic functionality but extreme performance requirement
A packet is routed based on the “Longest Prefix Match” (LPM) of its IP address with entries in a routing table.

Line rate and the order of arrival must be maintained.
Routing table: Sparse tree representation

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.14.7.3</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>10.18.200.5</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>7.14.7.2</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>5.13.7.2</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>10.18.200.7</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

⇒ 1 to 4 memory accesses
"C" version of LPM

```
int lpm (IPA ipa)
/* 3 memory lookups */
{
    int p;
    /* Level 0: 8 bits */
    p = RAM [ipa[31:24]]; if (isLeaf(p)) return value(p);
    /* Level 1: 8 bits */
    p = RAM [ipa[23:16]]; if (isLeaf(p)) return value(p);
    /* Level 2: 8 bits */
    p = RAM [ptr(p) + ipa [15:8]]; if (isLeaf(p)) return value(p);
    /* Level 3: 8 bits */
    p = RAM [ptr(p) + ipa [7:0]]; return value(p);
    /* must be a leaf */
}
```

Not obvious from the C code how to deal with
- memory latency
- pipelining

Must process a packet every 1/15 µs or 67 ns
Must sustain 4 memory dependent lookups in 67 ns

Real LPM algorithms are more complex
An implementation: Circular pipeline

The fifo holds the request while the memory access is in progress

in order
Is this design correct?

Does the look up produce the right answer?
- Easy: check it against the C program

Do answers come out in the right order?
- Difficult to state formally for designers
  - Is it even possible in a given logic?
- Alternative: The designer tags input messages and checks that the tags are produced in order
  - Challenge: verify this condition statically – can be done by someone other than the designer!
  - Code for tagging can be “erased” at any stage
Circular pipeline: A performance issue – Dead-cycles

Most performance concerns eventually become correctness concerns

>25% slowdown!

\[ \text{rule } \text{“enter” } y = \text{inQ.first()} \ \text{in} \]
\[ \text{(inQ.deq()) | mem.req(addr(y)) | fifo.enq(rest(y))} \]

\[ \text{rule } \text{“recirculate” } x = \text{mem.res()} \ \text{in} \ y = \text{fifo.first()} \ \text{in} \]
\[ \text{if } (!\text{done? (x)}) \ \text{then} \ ((\text{mem.deq()} ; \text{mem.req(addr(x,y))) |)} \]
\[ (\text{fifo.deq()} ; \text{fifo.enq(shift(y)))) \]
\[ \text{else} \ (\text{fifo.deq()} | \text{mem.deq()} | \text{outQ.enq(x)}) \]
Example 3:

**Lossy compression – H.264**

The standard only specifies decoding; encoder’s goal is to compress as much as possible without loss of perceptual fidelity
The standard is 400+ pages of English; the standard implementation is 80K lines of convoluted C. Each is incomplete!

Only viable correctness criterion is bit-level matching against the reference implementation on sample videos

Parallelization is more complicated than what one may guess based on the dataflow diagram because of data-dependencies and feedback

Errors don’t matter much
H.264 Decoder: Implementation

Different requirements for different environments
- QVGA 320x240p (30 fps)
- DVD 720x480p
- HD DVD 1280x720p (60-75 fps)

Each context requires a different amount of parallelism in different blocks
- Modular refinement is necessary
- Verifying the correctness of refinements requires traditional formal techniques (pipeline abstraction, etc.)
H.264: Bluespec Implementations

Productivity: Base profile
- Effort: Less than one-man year
- 8K lines of Bluespec (contrast 20k to 80K lines of C)
- First draft decoded 720p @ ~32fps, (Available C codes do not meet this performance)

Architectural Exploration: Many improvements made over a period of several months to increase performance and reduce area
- Process several samples / cycle
- Adjust FIFO depths
- Pipeline modules: Interpolator, Deblocking filter
- After improvements decodes 720p@ ~95fps and 1080p@ 70fps
- Area 4.4 mm sq (180nm)

Modular refinement is both feasible and essential
H.264 Takeaway

- Like most algorithmic designs bit-level matching against the C reference is totally acceptable
  - It’s not clear if formal techniques would generate any additional confidence in the design

- Modular refinement is extremely important in implementations
  - Formal techniques can be of significant value
Example 4:

Microprocessor design

Precise specifications are available and absolute correctness is required

The proving ground for all verification techniques
Microarchitecture of a modern CPU

- Speculative, out-of-order
- Many, many concurrent activities

Nirav Dave, MEMOCODE, 2004
ROB actions

**Re-Order Buffer**

<table>
<thead>
<tr>
<th>State</th>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>W</td>
<td>Instr A</td>
<td>V</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>Instr B</td>
<td>V</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>Instr C</td>
<td>V</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>Instr D</td>
<td>V</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>Instr -</td>
<td>V</td>
<td>V</td>
<td>-</td>
</tr>
</tbody>
</table>

**ALU Unit(s)**

- Get a ready ALU instr
  - Put ALU instr results in ROB

**MEM Unit(s)**

- Get a ready MEM instr
  - Put MEM instr results in ROB

**Register File**

**Decode Unit**

- Insert an instr into ROB
- Resolve branches

**Re-Order Buffer**

- Get operands for instr
- Writeback results
But, what about all the potential race conditions?

- Reading from the register file at the same time a separate instruction is writing back to the same location.
  - Which value to read?
- An instruction is being inserted into the ROB simultaneously to a dependent upstream instruction's result coming back from an ALU.
  - Put a tag or the value in the operand slot?
- An instruction is being inserted into the ROB simultaneously to a branch mis-prediction.
  - Must kill the mis-predicted instructions and restore a "consistent state" across many modules.

**Rule Atomicity**

- Lets you code each operation in isolation.
- Eliminates the nightmare of race conditions ("inconsistent state") under such complex concurrency conditions.

*All behaviors are explainable as a sequence of atomic actions on the state.*
Used 14 rules to describe an OOO processor and proved the correctness of the system with respect to the ISA.
Reactions (1997-98)

- **IBM, Intel**
  - Very interesting but engineers don’t prove theorems
  - Maximum money is spent on tools that work on implementation descriptions as opposed to models

- **Academics**
  - Not automated enough, no interesting decision procedures
"Automated" Processor Verification

- Models are abstracted from (real) designs
  - UCLID – Bryant (CMU) : OOO Processor hand translated into CLU logic (synthetic)
  - Cadence SMV - McMillian : Tomasulo Algorithm (hand written model. synthetic)
  - ACL – Jay Moore: (Translate into Lisp)
  - ...

- Some property of the manually abstracted model is verified
  - Great emphasis (and progress) on automated decision procedures

Since abstraction is not automated it is not clear what is being verified!

BAT[Manolios et al] is a move in the right direction
Example 5:

**Cache Coherence**

Dealing with nondeterministic specifications

Absolute correctness is required
Cache Coherence Protocols

- A memory model defines what a Load instruction can return from a possible set of Store values.
- Cache coherence (CC) protocols are designed to preserve a memory model in the presence of caches.
  - An efficient CC protocol is difficult to design; done by experts only.
    - Easy to make mistakes even in abstract protocols because of nondeterminism.
  - Lot of room to make mistakes even during the implementation of a correct protocol.
A personal anecdote

My student Xiaowei Shen designed an adaptive protocol called Caché

- Very difficult for me to understand and be sure of its correctness
- We used TRS to give a precise definition and TLAs to prove its correctness

This was not enough ...
Realization

- The proof was so long that we could have made a mistake easily
- Nobody else was going to read the proof – “it is not interesting”

Even though the protocol correctness is of extreme importance the burden of its correctness is solely on its designers.

*Different from a math theorem*

*Mechanical theorem proving*
It took Joe Stoy more than 6 months to learn PVS and show that some of the proofs in Xiaowei Shen’s thesis were correct.

This technology is not ready for design engineers.
Model Checking

- CC is one of the most popular applications of model checking
- The abstract protocol needs to be abstracted more to avoid state explosion
  - For example, only 3 CPUs, 2 addresses
- There is a separate burden of proof why the abstraction is correct
- Nevertheless model checking is a very useful debugging aid for the verification of abstract CC protocols
Implementation

- Design is expressed in some notation which is NOT used directly to generate an implementation
  - The problem of verification of the actual protocol remains formidable
  - Testing cannot uncover all bugs because of the huge non-deterministic space

Proving the correctness of cache coherence protocol implementations remains a challenging problem
Collective insights from these examples

- In some applications performance is paramount, even at the expense of some errors
  - 802.11, H.264

- In others, correctness is non-negotiable and performance is a secondary concern
  - IP look up, processors, caches

- Some properties are not easy to express formally but may be expressible by adding state to the design
  - Are packets processed in order?

- For some applications, specifications have to incorporate non-determinism
  - CC, speculative processors

Simulation and testing are here to stay *BUT* for commonly used refinements and transformations formal methods are preferable
Some common threads in current formal verification

- A manually abstracted model, as opposed to the real design (implementation) is verified.
- Great emphasis on automated decision procedures but not on automated model abstraction.
- Stating properties often has no direct benefit for the designer.

At the end of the day it is not clear if the formal verification of the model has any bearing on the verification of the real design.

Back to testing and debugging!
What is needed

- High-level notation
  - with precise semantics
  - capable of expressing nondeterminism
  - amenable to synthesis of actual implementation

- Powerful tools for proving properties of such designs

Automatic extraction of abstract models from designs expressed in Verilog or C or SystemC is a lost cause

Thanks!
Example 2:

An 802.11a Transceiver

Dealing with noise
802.11a Transmitter

Must produce one OFDM symbol (64 Complex Numbers) every 4 µsec

Uncoded bits

24

Controller

data

headers

Scrambler

Encoder

Interleaver

Mapper

IFFT

Cyclic Extend

accounts for 85% area
Verification Issues

Control is straightforward

- Small amounts of testing against the C code is sufficient, provided the arithmetic is implemented correctly
  - C code may have to be instrumented to capture the intermediate values in the FIFOs
- No corner cases in the computation in various blocks
  - High-confidence with a few correct packets

Still may be worthwhile proving that the (non standard) arithmetic library is implemented correctly
Modular refinement IFFT

Reuse the same circuit three times to reduce area
Circular pipeline: Reusing the Pipeline Stage
Superfolded circular pipeline: Just one Bfly-4 node!

Index: 0 to 15

Stage 0 to 2

4, 16-way Muxes

4, 16-way Demuxes

Index == 15?

64, 2-way Muxes
Modular Refinement - IFFT

When the designer changes the IFFT module, does the whole design have to be re-verified?
- Unit verification of the refined module should be sufficient provided the designer uses an appropriate design style.

Performance and area refinements in such algorithmic blocks are often based on standard transformations for parallelism and reuse.
- Formal verification of these transformations can almost completely eliminate the need for unit testing for such refinements.

“correct-by-construction”
802.11a transceiver:
Higher-level correctness

Does the receiver actually recover the full class of corrupted packets as defined in the standard?
- Designers totally ignore this issue
- This incorrectness is likely to have no impact on sales

Who would know?

If we really wanted to test for this, we could do it by generating the maximally-correctable corrupted traffic

All these are purely academic questions!
A performance issue

Are there any dead-cycles, i.e., can a new message enter the system in the cycle in which an old message leaves the system?

- The system will slow down more than 25% if there is a dead cycle.
- Formal verification?

Most performance concerns eventually become correctness concerns