What interactive theorem proving can do for Verilog hardware development

Andreas Lööw
Imperial College London
Summary

Claim 1:
Today you can do ITP-based development for software, this is useful

Claim 2:
If we had ITP-based development for hardware, it would be equally useful for hardware

... this talk is about adapting a development methodology from ITP-based software development for Verilog hardware development
Some ITPs/proof-assistants

Examples:
• Coq
• Isabelle/HOL
• HOL4
• Lean
• ACL2
• ...
Question: Why do *ITP*-based development instead of, e.g.: pen-and-paper mathematics, fully automated theorem proving, or something else?
ITP-based development – why

Main point of ITP:

- **Trustworthy proofs**, checked by small program ("kernel")

- Allows for **combining human and machine reasoning** – get the strengths of both, avoid the weaknesses of both

- Allows you to check/prove that **large developments “fit together”**
Non-ITP formal methods

- **Input**: Specification
- **Output**: Claim that program/circuit satisfies specification

- **Input**: Program/circuit description
- **Output**: Some low-level representation, e.g., machine code or netlists
Non-ITP formal methods

Verification tool

Input: Specification

Output: Claim that program/circuit satisfies specification

Compiler/synthesis tool

Input: Program/circuit description

Output: Some low-level representation, e.g. machine code or netlists

What we prove: correct

Actual end-result, what end-users get
Non-ITP formal methods

Output: Claim that program/circuit satisfies specification

Output: Some low-level representation, e.g. machine code or netlists

Input: Specification

Input: Program/circuit description

Important: The two tools must “interpret” the implementation language in the same way!

If only there were a way to ensure that they compose well... 😕
ITP formal methods

Verification tool
- Input: Specification
- Output: Claim that program/circuit satisfies specification

Compiler/synthesis tool
- Input: Program/circuit description
- Output: Some low-level representation, e.g., machine code or netlists
ITP formal methods

Verification tool:
- Input: Specification
- Output: Claim that program/circuit satisfies specification
- Correctness theorem: Program/circuit behaviour follows spec

Compiler/synthesis tool:
- Input: Program/circuit description
- Output: Some low-level representation, e.g., machine code or netlists
- Compiler theorem: Compiler is “semantics preserving”, i.e., output has same semantics/behaviour as input

Source-level correctness theorem + Compiler theorem = Target-level correctness theorem
ITP formal methods: Software

**Input:** Specification

**Output:** Claim that program/circuit satisfies specification

*Functional programming, program logic (e.g., SL), …*

**Input:** Program/circuit description

**Output:** Some low-level representation, e.g., machine code or netlists

*E.g., CompCert or CakeML*

**Input:** C or SML
ITP formal methods: Hardware

- **Input:** Specification
  - Functional programming, program logic (e.g., SL), ...

- **Output:** Claim that program/circuit satisfies specification

- **Input:** Program/circuit description
  - Verilog or VHDL

- **Output:** Some low-level representation, e.g. machine code or netlists
  - E.g., C or SML
  - E.g., CompCert or CakeML
  - Verilog synthesis tool: Lutsig

Problem: Our synthesis tool must be semantics preserving…
Verilog

Verilog and SystemVerilog Gotchas
101 Common Coding Errors and How to Avoid Them

Stuart Sutherland and Don Mills

Springer
Simulation-and-synthesis mismatches

Input: Specification

Output: Claim that program/circuit satisfies specification

Input: Program/circuit description

Output: Some low-level representation, e.g., machine code or netlists

Verilog or VHDL

Functional programming, program logic (e.g., SL), ...

Verilog synthesis tool

Verilog’s “simulation semantics”

Verilog’s “synthesis semantics”
Combinational logic -- mismatch example and handling it formally in Lutsig
“Mis-ordered” assignments

B.5 Assignment statements mis-ordered

```verilog
module andor1a(
    output logic y,
    input logic a, b, c);
logic tmp;

always_comb begin
    y = tmp | c;
    tmp = a & b; // write after read
end
endmodule
```
“Mis-ordered” assignment statements mis-ordered

module andor1a(
    output logic y,
    input logic a, b, c);
logic tmp;
always_comb begin
    y = tmp | c;
    tmp = a & b; // write after read
end
endmodule

Example from the “synthesis standard”

Essentially, a prose-specified event-driven operational semantics

There is an (stratified) event queue, handling of events, etc.

This block induces a software-like thread that will run each time something the block depends on change value

The statements run in the given order
B.5 Assignment statement mis-ordered

module andor1a(
    output logic y,
    input logic a, b, c);

logic tmp;

always_comb begin
    y = tmp | c;
    tmp = a & b; // write after read
end
endmodule

“Mis-ordered” assignments

“This standard defines a set of modeling rules for writing Verilog HDL descriptions for synthesis.”

“Combinational logic shall be modeled using [...] or an always statement.”
module andor1a(
    output logic y,
    input logic a, b, c);
    logic tmp;
always_comb begin
    y = tmp | c;
    tmp = a & b; // write after read
end
endmodule

Describes an event-driven language, could have equally well been a (weird) event-driven software programming language.

Gives us “modelling rules” for how to model/describe hardware.

Totally fine from the perspective of simulation, just propagate events as specified.

Makes no sense as a hardware model, sequential logic (stateful logic) inside block for combinational logic (stateless logic).
What happens when you give today’s synthesis tools a problematic design?

Basically anything, today’s synthesis tools might:

• abort (good case)

• emit warnings (borderline case)

• silently synthesise nonsense (bad case)

Such synthesis tools are not semantics preserving, i.e., this is bad.
Lutsig – a verified Verilog synthesis tool

• Developed and verified inside the HOL4 interactive theorem prover

• Designed to fit into ITP-based hardware development

• Specifically, semantics preserving

• (Can be used outside formal development as well, like any other synthesis tool.)
Lutsig – a verified Verilog synthesis tool

• Handles a small synthesisable subset of Verilog for synchronous designs

• Targets FPGAs:
  • Verified synthesis algorithm, based on open source CSYN synthesis tool
  • Translation-validation-based technology-mapping algorithm for FPGAs (LUTs)
  • Remaining steps outside formal development (e.g., P&R, bitstream encoding)
Lutsig’s correctness theorems (simplified)

Correctness w.r.t. (Lutsig’s) Verilog simulation semantics:

\[ \text{Lutsig}(D) = \text{OK}(N) \implies \forall n, \text{run}_\text{verilog}(D, n) = \text{run}_\text{netlist}(N, n) \]
(except for X-related behavior, which is allowed to be removed)

Correctness w.r.t. synthesis idiom for always_comb:

\[ \text{Lutsig}(D) = \text{OK}(N) \implies \forall \text{Verilog variables } v \text{ in } D, \]
if \( v \) written to by always_comb block \( \implies \)
no register with name \( v \) in netlist \( N \)
Lutsig vs. today’s synthesis tools

• Design your Verilog module using the old familiar synthesis idioms

• If Lutsig successfully gives back a synthesised netlist:
  • because of Lutsig’s correctness theorem, the synthesised netlist must have the same behaviour as the input Verilog module
  • i.e., simulation-and-synthesis mismatches are ruled out using mathematical proof

• If Lutsig errors out:
  • revisit your design
  • This happens e.g. when the simulation and synthesis semantics point in different directions, because Lutsig abides by both semantics, Lutsig is forced to abort if this happens
What does Lutsig actually do?

• Sequential blocks (always_ff) straightforward to handle, with check that blocking vs. nonblocking assignments are not misused

• Combinational blocks:
  • Sort blocks topologically w.r.t. read dependencies, e.g.:

    ```
    always_comb b = a + 1;
    always_comb a = inp;
    ```

  • (Abort if cannot sort.)

  • Examples of individual blocks to follow...
Combinational example 1: Scalars

For straight-line code, read as netlist:

```verbatim
always_comb begin
    // Lutsig would die here since tmp
    // read before written to
    y = tmp | c;
    tmp = a & b;
end
```

Cannot sort here since simulation semantics says to execute statements in order given!
Combinational example 2: Arrays

For straight-line code, read as netlist:

```verbatim
logic[1:0] foo;

always_comb begin
    foo[0] = inp1;
    foo[1] = inp2;
    // ok reading foo here since whole array covered
    foo = foo + 1;
end
```
Combinational example 3: If-statements

Generate mux for if-statements, fail if not assigned in all branches:

```vhdl
always_comb
    if (c)
        a = inp;
    //else
    //    a = 'x;
```
Remember: Lutsig is formally verified

• Previous slides are pretty much the same checks a helpful synthesis tool or a linter would do

• Lutsig, however, is formally verified

• So, we know that the checks done are sufficient to guarantee semantics-preserving synthesis, i.e., input Verilog module and output netlist behave the same
To do: Consider more sources of simulation-and-synthesis mismatches

- Obviously, need to go through the same process for other sources of simulation-and-synthesis mismatches

- SystemVerilog solves some things: e.g. incomplete sensitivity lists

- Some things should just be prohibited: e.g. delays

- Sometimes the simulation model is slightly off, e.g. dual-port block RAM
Conclusion

• Clearly, we want to do development inside ITPs

• Verilog is a... tricky language

• Lutsig is one attempt at doing formal hardware development using Verilog nevertheless

• Verilog is the most popular HDL, so it’s doing something right