Assertion Based Verification

Joe Hurd
joe.hurd@comlab.ox.ac.uk

CARD Group

Computing Laboratory
Oxford University

Contents

Assertion Based Verification

- Verification takes 70% of the hardware design cycle.
- Producing a specification of a circuit can be hard.
- Often much easier to write formulas (assertions) that describe error conditions.
  - **Testing** can simulate the circuit, and report a bug if an error occurs. This talk.
  - **Formal verification** can prove that these errors can never happen.
- This is Assertion Based Verification.
- Need a logic in which to write assertions.

PSL

- IBM's Sugar 2.0 language won a competition run by Accellera to find an industry standard assertion language.
- It was then renamed Accellera Property Specification Langauage (PSL).
- *"PSL is an intuitive, declarative language for describing behaviour over time."* [IBM]
- This talk: the Temporal Layer of PSL, essentially LTL with regular expressions:

Inside Temporal PSL

- Boolean Expressions
  - Evaluated on a single state.
- Sequential Extended Regular Expressions (SEREs)
  - Evaluated on a finite sequence of states.
- Foundation Language Formulas
  - Evaluated on a finite or infinite path of states.
  - This talk: will only consider infinite paths.

Verilog Checkers

- Suppose we have a circuit written in Verilog,
- and a PSL formula that we would like to hold of every simulation run of the circuit.
  - Think of a simulation run as an infinite path of states.
- We can code up the formula as a Verilog module that monitors the circuit.
  - But how to avoid bugs?
- Using HOL4, we can verify a translation from the PSL formula to a deterministic finite automaton.
  - The DFA is guaranteed to produce an error iff the PSL formula is violated on the simulation path.
  - Thanks to Mike Gordon's formalization of PSL.

Contents

Safety Violations

- Given a checking automaton for the PSL formula f,
- and an infinite path p,
- when can the automaton report a property violation?

$$\text{safety\_violation}\ p\ f \equiv \exists n.\ \forall q.\ |q| = \infty \Rightarrow \neg(p^{0,n}q \models f)$$

$$\underbrace{p_0 \quad p_1 \quad \cdots \quad p_n}_{\text{bad prefix}} \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \cdots \quad \models \neg f$$

- If the bad prefixes form a regular language, then we can detect safety violations with a finite state automaton.

Verified Checkers

- This is what we proved about checker automata:

$$\begin{aligned}\vdash \quad & \forall f, p. \\ & |p| = \infty \wedge \mathsf{simple}\ f \Rightarrow \\ & (\mathsf{safety\_violation}\ p\ f \iff \exists n.\ p^{0,n} \models \mathsf{checker}\ f)\end{aligned}$$

- checker maps a PSL *formula* to a PSL *SERE*.
- Not enough to have an implication, because otherwise a trivial checker $\top$ or $\bot$ would suffice.
- Condition 1: p is an infinite path.
- Condition 2: f is a simple formula.

Checkers: Next

- The next operator 'postpones' a formula by one step:

$$\vdash\ w \models \mathsf{next}\ f \iff |w| > 0\ \wedge\ w^1 \models f$$

- Next formulas are simple:

$$\vdash\ \forall f.\ \mathsf{simple}\ f \Rightarrow \mathsf{simple}\ (\mathsf{next}\ f)$$

- Next checkers just prepend the SERE $\{\top\}$:

$$\vdash\ \mathsf{checker}\ (\mathsf{next}\ f) = \{\top\}; \{\mathsf{checker}\ f\}$$

Checkers: Until

- The weak until operator is defined thus:

$$\vdash \quad w \models f \text{ until } g \iff$$
$$\forall j \in [0..|w|).\ w^j \models f \Rightarrow \exists k \in [0..j+1).\ w^k \models g$$

- The condition for weak until formulas to be simple:

$$\vdash \ \forall f, g.\ \text{simple } f \wedge \text{boolean } g \Rightarrow \text{simple } (f \text{ until } g)$$

- Weak until checkers are defined as

$$\vdash \quad \text{checker } (f \text{ until } g) \equiv$$
$$\{(\text{boolean\_checker } g)[*]\};$$
$$\{\{\text{checker f}\} \sqcap \{\text{boolean\_checker } g\}\}$$

Creating Verilog Checkers

- Take the SERE output of the checker, and lazily convert to a nondeterministic finite automaton (NFA).
- Compute the reachable states of the deterministic finite automaton (DFA) via transition theorems:

$$\begin{aligned} \vdash \quad & \forall c. \\ & \mathsf{REQ} \notin c \wedge \mathsf{ACK} \in c \Rightarrow \\ & \mathsf{transition}_D\ [6]\ c = [2;4] \end{aligned}$$

- Finally, print the whole DFA as a Verilog module.
  - An informal step, could introduce bugs :-(

Contents

Example: PSL Formula

From page 45 of the Accellera PSL Reference Manual:

$$c \wedge \text{next}\,(a \text{ until } b)$$

Their example actually uses strong until, we'll use weak until instead.

Example: SERE

```
|- checker (...example PSL formula...) =
   S_OR
     (S_BOOL (B_NOT (B_PROP c)),
      S_CAT
        (S_BOOL B_TRUE,
         S_CAT
           (S_REPEAT (S_BOOL (B_NOT (B_PROP b))),
            S_OR
              (S_AND
                 (S_BOOL (B_NOT (B_PROP a)),
                  S_CAT
                    (S_BOOL (B_NOT (B_PROP b)),
                     S_REPEAT (S_BOOL B_TRUE))),
               S_AND
                 (S_CAT
                    (S_BOOL (B_NOT (B_PROP a)),
                     S_REPEAT (S_BOOL B_TRUE)),
                  S_BOOL (B_NOT (B_PROP b))))))))
```

Example: Verilog Module

```
module Checker (a, b, c);
input         a, b, c;
reg    [2:0] state;
initial state = 0;
always @ (a or b or c)
begin
  case (state)
    0: if (c) state = 5; else state = 1;
    1: begin $display ("Checker: property violated!"); $finish; end
    2: begin $display ("Checker: property violated!"); $finish; end
    3: state = 3;
    4: if (a) if (b) state = 3; else state = 4;
       else if (b) state = 3; else state = 2;
    5: if (a) if (b) state = 3; else state = 4;
       else if (b) state = 3; else state = 2;
    default: begin $display ("Checker: unknown state"); $finish; end
  endcase
end
endmodule
```

Contents

- Introduction
- Verified Checkers
- An Example Formula
- **Conclusion**

Conclusion

- An interesting exercise that covers a wide range of formulas while staying within PSL.
- Real world applications of the Verilog checkers?
  - Require (verified) state minimization to be practical.
- Future Work: To extend our coverage, must drop SEREs as intermediate language.
  - Would like to implement weak suffix implication $\{\cdot\} \mapsto \{\cdot\}$ which is in the Accellera simple subset.