Deadlock Verification For Dummies – The Easy Way of Using SVA and Formal

Mark Eslinger Jeremy Levitt Joe Hupcey III
Mentor, A Siemens Business, Fremont, CA

Abstract

RTL simulation cannot directly tell if a digital system is deadlocked — you can only observe that nothing has happened for a long time, and this is highly dependent on the "right" stimulus being applied.

In contrast, Formal verification has the ability to find deadlock conditions in your design. However, the traditional iterative approach using written liveness and safety properties in combination with manually written constraints can be time consuming and error prone even in expert hands. While there are nonstandard assertion languages that can be used, these are reserved for academic practitioners and not useful for the typical RTL-aware design and verification engineers who use industry standard System Verilog Assertions (SVA).

In this paper we will show how combining the above concepts using normal SVA liveness properties allows for RTL engineers to achieve the benefit of formal deadlock analysis without the iterative component or learning a non-standard assertion language. Deadlock verification for dummies!

Two Deadlock Cases

A. Can your design get into a state from which it can never escape?

This is analogous to the trapped warehouse worker mentioned in the paper

B. Can your design get into a state in which you can stay as long as you like (by avoiding opportunities to escape)?

This is analogous to the couch potato mentioned in the paper

Limitations of Simulation

System deadlock is virtually impossible to detect with RTL VHDL or Verilog simulations!

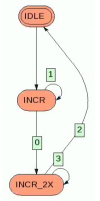
- Simulation cannot directly detect if the design is deadlocked
  - Can only observe that nothing's happened for a long time
  - How long is too long?
- Simulation cannot differentiate between cases A and B
  - True system lockup vs. potentially poor stimulus
- Simulation is dependent on users generating the "right" stimulus
  - This of course is how all bugs are missed with simulation, but particularly so for bugs that require a number of specific, synchronized interactions

Simple FSM Example

```
module dut (input logic clk, rstn, [1:0] din,
output logic [3:0] cnt);
  typedef enum logic [1:0] { IDLE, INCR,
  INCR_2X } State;
  State st;

 always_ff @(posedge clk or negedge rstn)
  if (~rstn) cnt <= 0;
  else cnt <= (st == IDLE)    ? cnt :
              (st == INCR)    ? cnt + din :
             /* st == INCR_2X */ cnt + 2*din ;

always_ff @(posedge clk or
            negedge rstn )
 if (~rstn) st <= IDLE;
 else
  case (st)
    IDLE: st <= INCR;
    INCR: st <= (din == 2'b10) ?
                INCR_2X : INCR;
    INCR_2X: st <= (cnt == 0) ?
                IDLE : INCR_2X;
  endcase // case (st)
endmodule // dut
```



Two Foundational Formal Concepts

"Safety"

Formally prove something bad will never happen

"Liveness"

Formally prove something good will eventually happen

Traditionally formal verification engineers would manually try to employ these to verify deadlock.
**** Requires expertise; tedious and unreliable ****

Translate Deadlock Into Formal Properties

A. Not expressible with either SVA or PSL
   – Can be described with computational tree logic (CTL)

```
AG EF (design_state !=
`SOME_PARTICULAR_STATE)
```

× CTL is too "academic" for regular engineers to use ☹

× CTL is not supported by commercial tools ☹

B. Standard SVA liveness property
   – Linear Temporal Logic (LTL) semantics

```
assert property (@posedge clk)
s_eventually (design_state !=
`SOME_PARTICULAR_STATE);
```

SVA Deadlock Properties

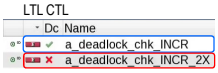
```
assert property (@posedge clk)
s_eventually (design_state !=
`SOME_PARTICULAR_STATE);
```

- Good for proving the absence of deadlock
  - If it is not possible to stay in a particular state even when you want to, it's certainly not possible to get stuck in that state
- Troublesome for finding deadlock
  - Will first find (many) Case B examples (escape is actually possible), before finding Case A (unescapable traps)
  - Must constrain away each Case B occurrence, and iterate again
    - Tedious and, depending on number of iterations, not necessarily practical

New Approach: Combine CTL/LTL Results

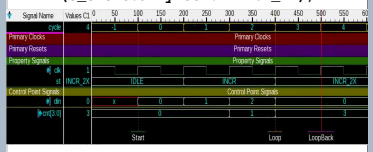
- **CTL Analysis**
  - ✓ Infer CTL property from SVA description
    No need for engineers to write CTL properties
  - ✓ Directly target "real" deadlock situations
    Enabled by new & improved formal engines
- **LTL Analysis**
  - ✓ Continuity with existing tool behavior
  - ✓ Leverage CTL analysis to expose escape routes

Results shown in easy to understand format



Case A Property and CEX (FSM Example)

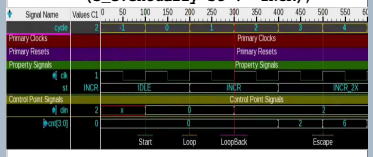
```
a_deadlock_chk_INCR_2X: assert property
(s_eventually st != INCR_2X);
```



In the above waveform, if the value of "cnt" is odd when the FSM transitions to the INCR_2X state, the FSM is in true deadlock since "cnt" can never equal 0.

Case B Property and CEX (FSM Example)

```
a_deadlock_chk_INCR: assert property
(s_eventually st != INCR);
```



In the above waveform, the FSM state INCR is deadlocked if the input "din" never equals 2. The escape path is when the input "din" equals 2. Constraining the input to eventually be that value would result in a proof, showing no deadlock is possible for the FSM state INCR.

Some CTL / Case A Complications

CTL deadlock counterexample

(once in a *loop* state it is not possible to ever see "Ack", regardless of inputs)

- Good for finding "real" deadlocks
- Not so good for proving there are no deadlocks
  - Can prove that there are no deadlocks to be found
  - But, proof does not necessarily mean that your system is deadlock-free !?
  - (This is why SVA/PSL are based on LTL and not CTL!)

Proof of Case (A) != deadlock-free

- Here is the catch
  - Addition of constraints can expose type-A deadlocks in a system that does not otherwise have them
  - Simple example: Reset -- if the design can always be reset, then type-A deadlock is not possible – asserting reset is always an escape option
- Case (A) is for **bug hunting**
  - Makes it much easier to find deadlocks
    - If you find a counterexample, you have deadlock in your design
  - But, bugs might be hidden due to *missing* constraints

Solution: Simultaneously Leverage Case (B)

LTL deadlock counterexample

(could loop forever, but might also be able to exit and see "Ack")

- If Case (A) is proven, case (B) counterexample (CEX) will be shown with escape routes
  - There must be an escape route, otherwise CEX would be a case (A) CEX
- Add constraints on the escape routes
  - Example: if reset used to escape, constrain reset to not assert after design initialization (normally done automatically)
  - Poster FSM Example: Constrain "din" to eventually equal the value of 2
- For a complete proof, iterate thru escape routes and add new constraints until:
  - There are no more type-B CEX – congrats, your system is deadlock-free!

OR

  - There IS a type-A CEX – meaning there is deadlock situation, and you have a CEX from formal analysis to debug/correct it

Example Deadlock Properties

Properties typically take one of two forms:

```
s_eventually(condition)
gating |-> s_eventually(condition)
```

req/ack type:

```
s_eventually(ack)    or
mode == `READ && no_intr && req
|-> s_eventually(ack)
```

Bus type:

```
s_eventually(pready)      (APB4)
s_eventually(awready)     (AXI4 AW)
```

Arbiter type:

```
s_eventually(~gnt[0])
```

```
Note: Full property syntax not shown
```

Find System Deadlock Issues Faster

- This is literally what formal was invented for!
  - Simulation is uncertain and inefficient in comparison
- Escape waveforms simplify analysis of LTL counterexamples
  - Missing constraints easier/faster to identify
- Typically, the number of illegal escape paths is small
  - Once these are constrained away, CTL-analysis is focused on finding bugs
  - For formal experts: CTL-analysis does not need "fairness constraints"

Summary

- The risk of a design going into deadlock is nearly impossible to detect with RTL simulation; hard to do with traditional formal
- Combining "LTL" and "CTL" analysis results, leveraging standard SVA syntax, and using new & more powerful CTL engines, enables regular engineers to effectively utilize CTL analysis
- Detecting RTL deadlock is now easier with Mentor's PropCheck using these advanced algorithms under-the-hood