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

Mark Eslinger[1] Jeremy Levitt[2] Joe Hupcey III[3]

[1]Mentor, A Siemens Business, 46871 Bayside Pkwy, Freemont, CA 94538, Mark_Eslinger@mentor.com
[2]Mentor, A Siemens Business, 46871 Bayside Pkwy, Freemont, CA 94538, Jeremy_Levitt@mentor.com
[3]Mentor, A Siemens Business, 46871 Bayside Pkwy, Freemont, CA 94538, Joe_Hupcey@mentor.com

***Abstract*- RTL simulation cannot directly tell if a digital system is deadlocked — you can only observe that nothing has happened for a long time (so when should you get worried that nothing is happening?), and you cannot can't differentiate between situations where your system is truly locked up from a situation where the right stimulus hasn't come along to take the design out of a local minimum. Additionally, simulation is dependent on the engineer knowing the right stimulus to trigger a particular problem. 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 non-standard 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!**

I. Introduction

The Dining Philosophers Problem[1] has long been used as a mechanism in computer science and other disciplines for teaching deadlock and various mechanisms for preventing it. The problem definition has 5 silent philosophers sitting around a table. In front of them are plates of spaghetti and a fork on one side and a spoon on the other. The philosophers can either eat or think. In order to eat, they need both a spoon and a fork. The philosopher can eat or think for as long as they like. The challenge is to develop a solution such that no philosopher starves. One issue might be that the philosopher switches hands with the fork and spoon so another philosopher would have both of either which is no good. Hence the development of the spork! A classic deadlock scenario in this case is all philosophers have a spork in their left hand, waiting for someone to put theirs down so someone can eat. In short, this case is a model for system deadlock when arbitrating between limited shared resources, and is something that happens in various scenarios in common digital designs.

The problem as defined by system deadlock or lockup is notoriously difficult to detect with RTL VHDL or Verilog simulations. However, exhaustive formal-based analysis is uniquely qualified to deliver results in this domain. Specifically, deadlock properties – concise, human and machine readable descriptions of the desired design behavior you want to verify – have been extensively studied and can be precisely specified using mathematical languages such as Linear-Temporal Logic[2] (LTL) or Computational-Tree Logic[3] (CTL).

Unfortunately, LTL and CTL are academic constructs, and as such are too cumbersome and lack support for use in industrial verification. Alternatively, formal verification engineers often try to leverage two foundational formal analyses supported by most commercial verification tools: "liveness" (formally prove something good will eventually happen) and "safety" (formally prove something bad will never happen). Unfortunately the expertise required to manually effectively combine these for deadlock verification is substantial, and even in the hands of an expert the approach is error prone.

Hence, this paper discusses the application of new automation that leverages familiar, industry-standard System Verilog Assertion (SVA) code to specify constraints and properties to detect deadlock in RTL designs – while still leveraging the concepts behind LTL, CTL, liveness, and safety analyses under-the-hood.

II. Deadlock Concepts

There are two primary deadlock scenarios or cases to consider.

Case A: Can your design get into a state from which it can never escape?
Case B: Can your design get into a state from which you can stay as long as you like by avoiding opportunities to escape?

Another way to look at this is the concept of the warehouse worker and the couch potato. When the worker went into one of the storage rooms in the warehouse, they got trapped inside the room when the door slammed shut and triggered a poorly designed locking mechanism to close. Now if the worker is lucky, there is a sledgehammer in the store room they can use to smash the lock. In the digital realm this may be akin to hitting the design with the heavy hammer known as reset! In most cases though we want to take that off the table from an analysis perspective or consideration in this case as well. In a nutshell, this is Case A above.

The other scenario is a person sitting on their couch watching football or playing a video game with all the snacks and drinks they like available to them. They could leave the couch and go do something outside but the game is so entertaining and the food so good that they decide to stay there indefinitely. Someone else may force them to leave, there are options to do so, however the possibility to stay a long time is there and they take it unless acted on by some outside force. This is equivalent to Case B above.

Verification of these scenarios in your digital design with simulation is virtually impossible. Simulation can't do deadlock verification in an efficient or complete manner. There are 3 primary reasons for this:

1) Simulation can't directly detect if the design is deadlocked.
2) Simulation can't differentiate between Case A and Case B.
3) Simulation is dependent on users generating the "right" stimulus.

For item 1) above, the only thing that can be observed is that some state is in the same state for some long period of time. Who determines what a "long" time is? How long is too long? For item 2) above, is the condition true lockup or just poor stimulus? This relates to item 3) above. The user has to generate the "right" stimulus to get into the deadlock state in the first place. Once in that deadlock state, is there some stimulus that will allow for that state to be escaped? Coming up with any of the above stimulus is time consuming and most likely incomplete. The typical verification engineer doesn't have time for it. Hence you are finding things in a hit or miss fashion. Enter formal verification!

Because it employs an exhaustive, mathematical analysis, formal verification is uniquely qualified to detect the risk of a design going into deadlock. Traditional formal-based approaches use two foundational formal analysis concepts: liveness and safety. *Liveness* properties are used to specify that something good will eventually happen. *Safety* properties state that something bad will never happen. The safety property has a counter example that shows the bad thing happening as shown below in Figure 1.

Figure 1: Safety property counter example, showing an ack without a req (bad thing happens)

The liveness property has a counter example that shows a good thing never happening. This is showing a deadlock scenario. It exhibits itself as a loop the design gets stuck in as shown below in Figure 2.

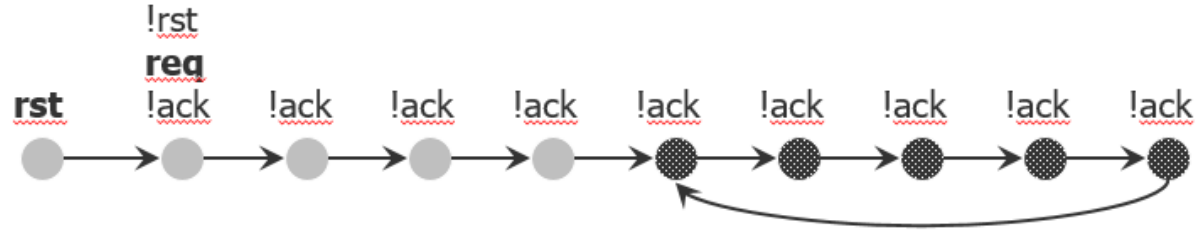

Figure 2: Liveness property counter example, showing deadlock, where the ack never shows up for the req

Traditionally formal verification engineers would try to manually employ these to verify deadlock. The challenge is this requires expertise and is tedious and unreliable. An example method using traditional SVA safety properties to check for deadlock is shown below:

1) Find a trace where the system remains in `SOME_SPECIFIC_STATE for N cycles:

```
cov_deadlock: cover property (@(posedge clk)
                    (design_state == `SOME_SPECIFIC_STATE)[*30]);
```

2) Using that trace to initialize the design, test if that trace happens to be a deadlock scenario:

```
a_test_deadlock: assert property (@(posedge clk) design_state == `SOME_SPECIFIC_STATE);
```

3) If deadlock not found, modify the cover property to find a different scenario and repeat

The challenge with the above is what values to try for N. You get deeper and if you find a proof, then how do you debug it? You need counter examples for useful debug. If you have to repeat step 3 over and over it isn't practical or fun! With formal, you can use liveness properties to verify deadlock. However, due to the semantics used this isn't complete as well. Taking the 2 cases discussed above with the worker and the couch potato:

Case A: This can't be verified with SVA or PSL. This can be done with CTL which no user wants to deal with!

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

What does AG and EF mean? Layman's english translation:

There is always a way to exit `SOME_PARTICULAR_STATE, no matter how you got there.

Case B: This can be done with a normal SVA liveness property (which uses LTL semantics)

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

So a user can write properties for Case B in today's tools. However, this isn't showing you true deadlock so there is a lot of onion peeling involved. To show true deadlock you need the CTL version or Case A. Questa PropCheck[4] has the ability to allow the user to write the property for Case B and also do the Case A analysis to give the full deadlock picture such that when you get a deadlock counter example, it is truly showing a deadlock condition in your design.

.

III. Deadlock analysis Considerations

This formal technique can be used at any development stage – from bring-up to bug hunting; and can be combined with any formal setup/environment, including abstractions. In contrast, simulation usually needs to create specific configurations or add watchdogs with arbitrary maximum values on specific "events".

Overall, there are primary lessons here:

1 – The risk of a design going into deadlock is nearly impossible to detect with RTL simulation, but is relatively easy with formal property checking

2 – SVA "liveness" properties are restricted to linear-temporal logic (LTL), whereas a deadlock check is a computational-tree logic (CTL) property.

3 – LTL properties are still valuable: LTL CEXs can expose missing setup constraints for the CTL property; CTL deadlock checks can be usefully inferred from existing SVA LTL properties.

If situation A occurs, the system design clearly is deadlocked. However, if situation A is "proven" to not occur, it does not necessarily follow that your system is deadlock-free. The catch is that the addition of constraints can expose type-A deadlocks in a system that does not otherwise have them. A common example involves reset. If the design can always be reset, then type-A deadlock is not possible – asserting reset is always an escape option.

The fact that adding constraints can expose bugs is a new concept for verification engineers used to writing properties using industry standard SVA. For SVA (or any LTL-based language), constraints are added only to remove false failures (i.e. counterexample waveforms that rely on illegal input stimulus). However, for deadlock verification, correctly specifying constraints is key to finding system deadlocks.

This is where situation B is useful. Formal can identify reachable states which are possible to stay in indefinitely, and then also show any available escape routes. Specifically, the output waveform will have a stem showing where the deadlock state is reached; a loop, where the design stays stuck in the deadlock state; and then a tail, where the design escapes the deadlock state. If an escape route is invalid (e.g. asserting reset; a technically available option but not something you want the DUT's end-user to be forced to do all the time), that route can be constrained away (e.g.

constraining reset to not assert after design initialization). If the loop itself is invalid (e.g. the 'ack' signal is never asserted on the interface), then the loop can be constrained away (e.g. once a 'req' has been issued, an 'ack' eventually has to appear on the interface – sometimes this is called adding a "fairness constraint").

There is a process you can followed with this method which is fairly straightforward and makes use of both Case A and Case B scenarios:

1) If you have a Case A type counter example, you have deadlock in your design, you need to fix it.
2) If you have a Case A type proof, that doesn't mean your design is deadlock free, it still may be possible. Note: Case A is a bug hunting tool, finding a CEX means you have a bug, a proof isn't a guarantee there isn't a bug. Further constraint adding is needed.
3) If Case A is proven, a Case B counter example will show escape routes from the loop as shown below in Figure 3:

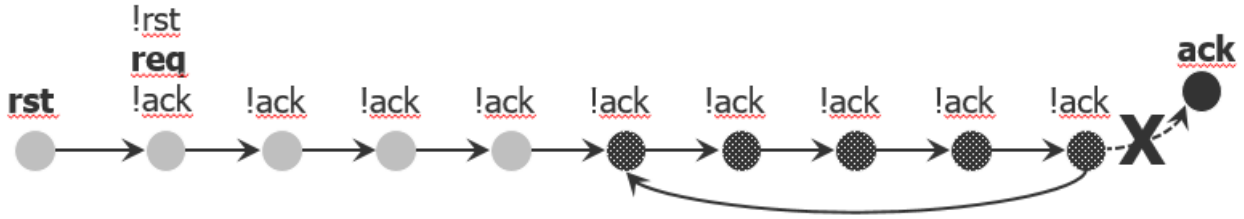

Figure 3: Case B counter example showing an escape route where the ack happens, constrain these away.

4) Start adding constraints to remove "invalid" escape routes. Examples of these would be constraining the reset to its inactive state or disabling testmode or scan enables.
5) Keep iterating on adding constraints until there are no more Case B counter examples in which case your design is deadlock-free (as long as your constraints are correct!) OR there is a Case A counter example meaning you have found a deadlock condition and there are relevant waveforms available to debug and fix the bug
6) Once all the found bugs are fixed you should have a full proof on your original liveness property in which case there is no deadlock.

This type of analysis is what formal was invented for! Allowing design and verification engineers in this case to find deadlock in their designs, if it exists. When going through the above process typically the number of illegal escape paths is small and if a deadlock scenario exists typically it is found after a few iterations if that. The benefit here as well is with this type of flow the CTL type checking that is being done doesn't need typical fairness constraints added to ensure an "ack" will show up on an input. The check is trying to do that for you! As with any formal analysis, any proof you get by adding constraints is of course dependent on the validity of the assumptions or constraints. These should be verified as you would do with any formal analysis. The power of this type of analysis is now available for formal verification engineers to use without the need to learn academic languages.

IV. SIMPLE EXAMPLE

To further demonstrate these ideas a simple example is used to make clear the difference between these two cases. Consider a design that has a counter and an FSM. There are 4 ports including clk, rstn, in[1:0], and cnt[3:0]. The simple FSM has the bubble diagram shown in Figure 4 below:

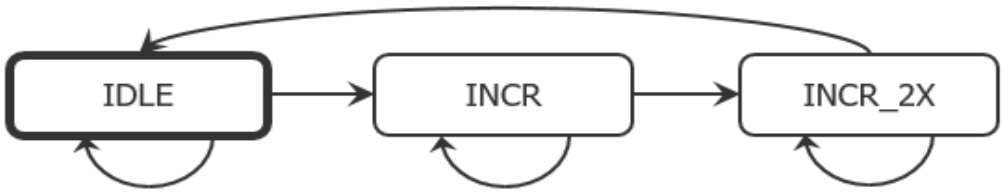

Figure 4: Simple FSM

The counter operates based on the states of the FSM as shown below:
FSM = IDLE => cnt retains its value
FSM = INCR => cnt increments by the value of the input
FSM = INCR_2X => cnt increments by 2x the value of the input

The FSM changes state based on the values of the input as well as the value of the counter:
FSM = IDLE: Remains in this state if the input is 0, otherwise move to state INCR
FSM = INCR: Remains in this state unless in the input = 2, then moves to state INCR_2X
FSM = INCR_2X: Remains in this state until the value of the counter = 0, then moves to state IDLE

The source code for the simple design is shown below in Figure 5:

```
module top (input logic clk, rstn, [1:0] in, output logic [3:0] cnt);

   typedef enum logic [1:0] { IDLE, INCR, INCR_2X } State;
   State cstate,nstate;

   always_ff @(posedge clk or negedge rstn)
     if (~rstn) cnt <= 4'h0;
     else cnt <= (cstate == IDLE)     ? cnt :
                 (cstate == INCR)     ? cnt + in :
               /* cstate == INCR_2X */ cnt + 2*in ;

   always_ff @(posedge clk or negedge rstn)
     if (~rstn) cstate <= IDLE;
     else   cstate <= nstate;

   always @*
       case (cstate)
         IDLE:    nstate <= (in != 2'b00) ? INCR    : IDLE;
         INCR:    nstate <= (in == 2'b10) ? INCR_2X : INCR;
         INCR_2X: nstate <= (cnt == 4'h0) ? IDLE    : INCR_2X;
         default: nstate <= IDLE;
       endcase

endmodule
```

Figure 5: Simple example rtl code to demonstrate deadlock cases for Case A and Case B.

Ultimately we'll check if the FSM states can become deadlocked. This will require some properties which are shown below in Figure 6:

```
bind top deadlock_chk bind_top (.*);

module deadlock_chk (
input clk,
input rstn,
input [1:0] in,
input [3:0] cnt
);

typedef enum logic [1:0] { IDLE, INCR, INCR_2X } State;
State fsm;
assign fsm = top.cstate;

a_deadlock_idle:  assert property (@(posedge clk) s_eventually(fsm != IDLE) );
a_deadlock_incr:  assert property (@(posedge clk) s_eventually(fsm != INCR) );
a_deadlock_incr2: assert property (@(posedge clk) s_eventually(fsm != INCR_2X) );

endmodule
```

Figure 6: Deadlock properties for simple example design

In the above example, the results from the formal run would show 3 Firings or Counter Examples (CEX). Let's take each case in order with the assumption that toggling reset is off the table:

1) IDLE Deadlock: Case B deadlock, there is an escape path
2) INCR Deadlock: Case B deadlock, there is an escape path
3) INCR_2X Deadlock: Case A deadlock, truly deadlocked!

For item one, if the input stays at 0, the FSM is deadlocked in the IDLE state. In normal flows, the user would have to add a fairness constraint to say that eventually the input is non 0. In this method, an escape waveform is shown automatically where in the input is a 1 and hence the FSM transitions to the next state. The same type of scenario can be used for the INCR state analysis. In these two checks, the escape waveform is valid, so no new constraints are added. A user typically wants to jump to the case where there is true deadlock. If the Case B CEX are invalid, then adding constraints could show a true deadlock, or a proof. An escapable waveform is shown below in Figure 7 for the 2nd case showing how to escape the INCR state:

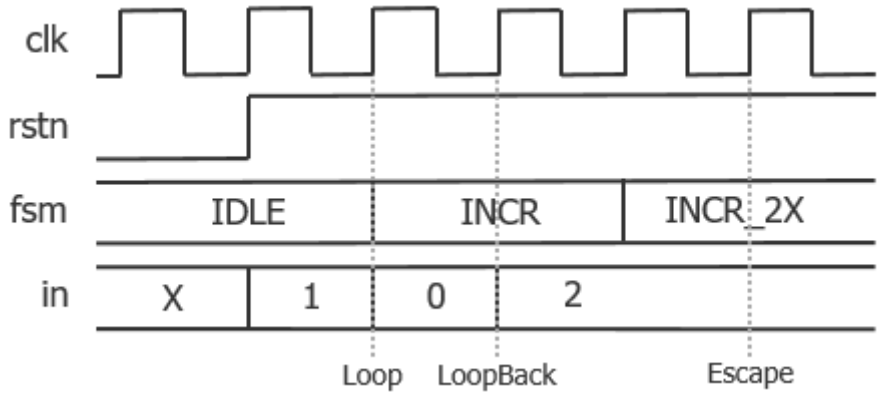

Figure 7: Waveform showing how to escape from the FSM=INCR deadlock condition (Case B Example)

The more interesting case is the true deadlock that happens for the INCR_2X FSM state. This state doesn't have to be deadlocked assuming the inputs are behaving in a certain way. However, when a certain sequence of events happens, then this state is truly deadlocked (minus the reset scenario which has been constrained off). Below in Figure 8 is a waveform showing the deadlocked state INCR_2X:

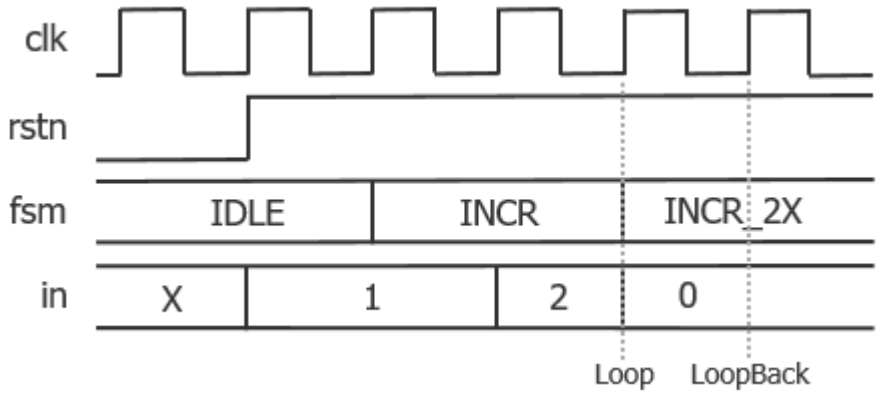

Figure 8: Waveform showing true deadlock for FSM=INCR_2X (Case A Example)

The reason for the deadlock in this design occurs when the counter is an odd number when the FSM enters the INCR_2X state. In this state the counter will always increment with an even number. Since the counter is starting from an odd state, when incremented with an even number, will never hit 0. Hence the FSM can't exit the INCR_2X state since that only occurs when the counter hits 0. A simple example that illustrates the differences between the Case A and Case B deadlock types. How does this apply in the real world?

V. COMMON EXAMPLES

There are a number of areas where these techniques can be applied for verification of system-level/architectural deadlock scenarios including FSM state as shown in the above example, req/ack type scenarios, bus interfaces, and arbitration to name a few. You are checking that some state will either happen like the presence of an acknowledge signal or not happen like the example above where the FSM isn't in a particular state. Typically properties will fall into a couple of categories:

1) Some gating conditions `|->` s_eventually(condition) e.g. `req |-> s_eventually(ack)`
2) s_eventually(condition) e.g. `s_eventually(ack)`

Interestingly enough you don't need to specify that the req happens to check if it is possible for the ack to show up. Formal will put the design into the state such that the check can happen. Also, with this type of analysis there isn't a need to add a fairness constraint on the ack signal. If you look at the properties from the simple example above you will notice they are checking that the design isn't in some state. The formal tool will put the design into that state to then check that it can exit that state. This can make things easier to describe. In the cases where implication is used it more often is used to specify some qualifying condition, for example:

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

So, to verify whether an acknowledgement signal ('ack') will ever follow a request ('req'), the user would apply the following assertion to drive the deadlock analysis. Another example showing a clear for a req is also shown:

```
a_ack_deadlock: assert property (@(posedge clk) s_eventually(ack) );

q_req1_clr1_deadlock: assert property (@(posedge clk) disable iff (!rstn)
                          flow_req[1] |-> s_eventually(flow_req_clr[1]) );
```

Similarly, to verify that an arbiter will release a given grant and not stay deadlocked in a grant state, this property would be analyzed against the DUT:

```
a_arb0_deadlock: assert property (@(posedge clk) s_eventually(~gnt[0]) );
```

This property would be repeated for all the bits of the "gnt" signal.

To verify that a bus interface, say APB4 for example, will have a ready signal and not stay deadlocked in a transaction, this property would be analyzed against the DUT:

```
a_apb4_deadlock: assert property (@(posedge pclk) s_eventually(pready) );
```

Generally for this type of check there wouldn't be preconditions added. However in the case of an inconclusive the check could be split up into a read and write version to further constrain the analysis:

```
a_apb4_wr_deadlock: assert property (@(posedge pclk)
         pselx && penable &&  pwrite && !pready |-> s_eventually(pready) );

a_apb4_rd_deadlock: assert property (@(posedge pclk)
         pselx && penable && !pwrite && !pready |-> s_eventually(pready) );
```

Similarly to verify that a bus interface like the AXI4 slave address/data channels, will have a ready and valid signals and not stay deadlocked in a transaction, these properties could be analyzed against the DUT:

```
a_axi4_aw_deadlock: assert property (@(posedge aclk) disable iff (!aresetn)
                                    s_eventually(awready) );
```

You would write similar properties for the arready, wready, bvalid, and rvalid signals.
Note: For interface type checks the user can write the assertions either on the master side or the slave side.

For cache type designs there are a number of properties that can be written. Some may look similar to the above types. For example a snoop req/ack type scenario:

```
a_snoop_req_ack_deadlock: assert property (@(posedge clk)
                    l2_l1snoopreq && (l2_l1snoopid == 2'b01) |->
                    s_eventually(l1_l2snoopack && (l1_l2snoopid == 2'b01)) );
```

The state of multiple caches can also be checked from a snooping perspective (modeling code not shown)

```
a_cache_hit0_invalid2_deadlock: assert property (@(posedge clk)
                  wr_hit0 && shared0 && wr_hit2 |-> s_eventually(invalid2) );
```

The above properties are liveness properties which are typically harder to solve than the standard safety properties commonly used. It is beyond the scope of this paper to go into various techniques used by formal to simplify the problem to get a result from the formal run. One thing that can be done is to break properties up into smaller sections of a flow such that it is easier to solve each individual one. An example is given below:

```
a_mtxdat_last_deadlock: assert property (@(posedge clk) mtxdat_act |->
                                  s_eventually(mtxdat_act && mtxdat_last) );

a_mtxdat_rdy_deadlock: assert property (@(posedge clk) mtxdat_act |->
                                                s_eventually(mtxdat_ready) );
```

In all of the above cases or other cases where it would make sense to apply deadlock verification, it is recommended that the user verify the design for correct operation before applying these deadlock techniques. Normally a user would run formal using safety assertions as well as verification IP for checking bus interfaces to wring out any function bugs in the design. For example you would want your AXI or APB interfaces to be functioning correctly before checking for deadlock scenarios, your FSM's behaving correctly, your arbiters able to arbitrate on each request/grant correctly, request/acknowledge handshake interfaces to function correctly, and cache related or other scenarios you may want to apply this to functioning correctly at some baseline level. Deadlock verification involves liveness properties which are typically harder to solve than safety properties. You want your analysis focused on deadlock scenarios instead of normal functional bugs. When the design is behaving correctly from a normal operational perspective, the more likely it is you will uncover the more complex deadlock corner case scenario bugs if they exist.

VI. Conclusion

The risk of a design going into deadlock is not something you can effectively verify with simulation. You could try to do it with traditional formal liveness and safety properties, but that can be a painful and error prone process. Fortunately, the esoteric LTL and CTL algorithms have been embedded in Mentor's PropCheck formal property checking tools to help you detect deadlock scenarios and uncover hidden escape routes. This innovative solution allows verification engineers to extend the traditional SVA syntax and quickly differentiate between a Case A and Case B type deadlock scenarios as described in this paper.

References

[1] E. Dijkstra and T. Hoare, "Dining philosophers problem", https://en.wikipedia.org/wiki/Dining_philosophers_problem
[2] A. Pnueli, "Linear temporal logic", https://en.wikipedia.org/wiki/Linear_temporal_logic
[3] E. Clarke and E. Emerson, "Computation tree logic", https://en.wikipedia.org/wiki/Computation_tree_logic
[4] Questa PropCheck User Guide, Mentor, A Siemens Business, November 2019.