2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Using Formal to Prevent Deadlocks

Abdelouahab Ayari[1] Mark Eslinger[2] Joe Hupcey III[3]

[1]Mentor, A Siemens Business, Munich, Germany, Abdelouahab_Ayari@mentor.com
[2]Mentor, A Siemens Business, Fremont, CA U.S.A., Mark_Eslinger@mentor.com
[3]Mentor, A Siemens Business, Fremont, CA U.S.A., Joe_Hupcey@mentor.com

***Abstract*- System deadlock is very difficult to detect with RTL VHDL or Verilog simulations. In fact these are the most difficult type of bugs to find! At best, you monitor key signal(s) to see if they have not changed for a long period of time -- but how long should you observe these signals before concluding that deadlock will not occur? Historically the solution has been to check the signal(s) with local and global watchdog timers; which can be hard to cleanly include. The truth is that with simulation-based approaches you can't differentiate between situations where your system is truly locked up vs. a case where the right stimulus hasn't come along to move the design to the next state. Additionally, simulation is dependent on the designer knowing the "right stimulus" to trigger a specific deadlock scenario. In contrast, Formal analysis has the ability to exhaustively find deadlock scenarios 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 for formal experts. While there are academic assertion languages that can be used, these are designed for academic practitioners and are not useful for RTL-aware design and verification engineers who use industry-standard System Verilog Assertions (SVA). Hence, in this paper we will show how using normal SVA liveness properties along with academic concepts allows for RTL engineers to achieve the benefit of formal deadlock analysis without the iterative component or learning a non-standard assertion language. This simplifies the process of finding and preventing deadlock in your design.**

I. INTRODUCTION

The Dining Philosophers Problem[1] is a classic example for teaching how deadlocks can occur in systems with shared resources, and for discussing the 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 can commonly occur in digital designs where some sort of arbitration is needed to enable access to shared resources.

The problem as defined by system deadlock/lockup is notoriously difficult to detect with RTL simulation. Formal-based analysis which is by nature exhaustive is uniquely qualified to deliver results in this domain. Deadlock properties 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).

LTL and CTL are academic constructs, and unfortunately, as such are too cumbersome and lack support for use in industrial verification. As an alternative, formal verification engineers often try to leverage two foundational formal analysis supported by most commercial formal 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. BASIC DEADLOCK CONCEPTS

There are two deadlock scenarios that must be considered:

Case A: Is it possible for your design to get into a state from which it can never escape?

Case B: Is it possible for your design to get into a state from which it can stay as long as it likes by avoiding opportunities to escape?

A simple way to look at this is an example from the animal kingdom. Sometimes animal control will need to come and trap critters that are causing damage. An example of this is an animal that has been caught in a trap from which it cannot escape. This is an example of Case A above.

The other case can be illustrated by an animal that has found a comfortable resting spot with plenty of food and water, but it is free to leave at any time. However, since the spot is so accommodating that the animal chooses to stay there indefinitely. This is an example of Case B above.

Simulating these deadlock scenarios in your design is nearly impossible for the following reasons:

1) A user can't directly detect if the design is deadlocked with simulation.
2) A user can't tell the difference between Case A and Case B with simulation.
3) A user has to supply the "right" stimulus to find deadlock with simulation.

The only thing that can be observed in simulation is that the DUT has been in a certain state for a long period of time. In the past watchdog timers have been used to extract the design from what might be a deadlock state. This isn't the most ideal implementation method given the latency, power, area, etc. this can introduce. Besides, the question remains, is a given "time out" condition a true lockup or just poor stimulus? To find out, the user has to generate the "right" stimulus to show the deadlock scenario. This is very difficult in the first place as there are millions of possibilities. If a potential deadlock state is found, once in this state is there some stimulus that can show an escape path? Generating the above stimulus is time consuming and most likely incomplete. Verification engineers don't have time to guess at what might work. This is where formal verification excels!

Formal uses an exhaustive, mathematical analysis, and is thus uniquely qualified to detect that a design can go into deadlock. There are two formal analysis concepts: liveness and safety. *Liveness* properties are used to specify that something good will eventually happen. *Safety* properties are used to specify that something bad will never happen. The safety property has a counter example (CEX) that shows a bad scenario happening as shown below in Figure 1.

Figure 1: Safety property CEX, showing an 'ack' without a 'req'

The liveness property has a CEX that shows a good thing never happening. The CEX shows the deadlock scenario as a loop the design gets stuck in as shown below in Figure 2.

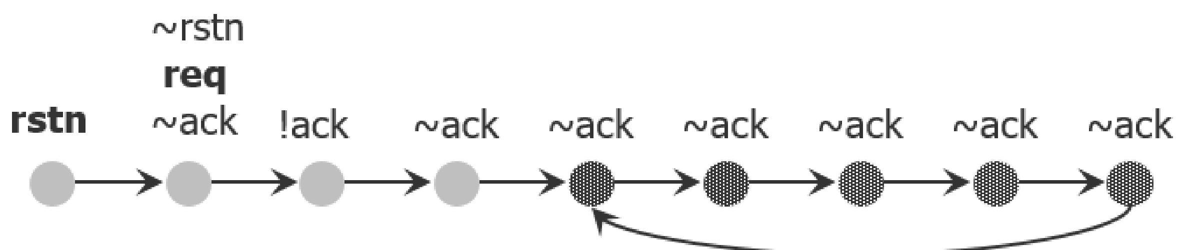

Figure 2: Liveness property CEX, showing deadlock, where the 'ack' never shows up for the 'req'

In the past formal verification engineers would use such properties to verify deadlock. However, this requires expertise with formal analysis; and like many manual techniques, it is tedious and unreliable. The following is an example of this traditional method using SVA safety properties to check for deadlock:

1) Find a trace where the design remains in `SOME_SPECIFIC_STATE for M cycles:
```
trace_deadlock: cover property (@(posedge clk)
                          (reg == `SOME_SPECIFIC_STATE)[*50]);
```
2) Use that trace to initialize the design and then test if that trace happens to be a deadlock scenario:
```
a_chk_deadlock: assert property (@(posedge clk)
                           reg == `SOME_SPECIFIC_STATE);
```
3) If property not proven(deadlock), modify the cover property and find a different trace and repeat

The difficult thing with the above is what values to try for M. If you find a proof, then how do you debug the deadlock? You need a CEX to properly debug a deadlock. Typically you have to repeat step 3 over and over which 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. Taking the 2 cases discussed above:

Case A: This can't be verified with SVA or PSL. This can be done with CTL, an academic language.
```
AG EF (reg != `SOME_PARTICULAR_STATE)
```
What is the definition of AG and EF? The meaning of the property is described below:
There is always a way to exit `SOME_PARTICULAR_STATE, no matter how you got there.

Case B: This can be described with a SVA liveness property (LTL semantics)
```
a_chk_deadlock: assert property (@(posedge clk)
            s_eventually(reg != `SOME_PARTICULAR_STATE );
```

A person can write a property for Case B and run it in current formal tools. However, this doesn't show true deadlock so there are a lot of iterations as described above. To show true deadlock you need the CTL version or Case A. Questa PropCheck[4] natively supports the writing of properties for Case B analysis that are effectively extended to simultaneously perform the Case A analysis. Hence, when PropCheck outputs a deadlock counter example, it is showing you a real deadlock scenario in your design.

III. Some Deadlock analysis Considerations

The formal deadlock verification technique can be used at any point during your verification flow and can be combined with any formal setup/environment, including abstractions. As mentioned above, simulation typically requires you to create specific configurations/scenarios or add watchdogs with arbitrary maximum values on specific states of the design.

Let's review what you have learned so far:

1 – The chance of a design going into deadlock is virtually impossible to detect with RTL simulation, and is relatively easy with formal property checking

2 – SVA "liveness" properties semantics use linear-temporal logic (LTL), and a true deadlock check requires a computational-tree logic (CTL) type property.

3 – LTL properties are valuable in that a LTL CEX can expose missing setup constraints for the CTL property and CTL deadlock checks can be inferred from existing SVA LTL properties.

When you run with this deadlock checking flow, the SVA liveness property will run with both LTL and CTL semantics. It will give you two types of results[5]: maybe-escapable and unescapable.

	Proven: not maybe-escapable	**Maybe-escapable**
Proven: not unescapable	No deadlock!	Escapable deadlock – **examine it**
Unescapable	-	Real deadlock – **design bug**

Elaborating on the table above: if the LTL property is proven there is no deadlock. If the LTL property fires then you look at the CTL results. If the CTL result has a firing there is a true deadlock scenario that can be debugged. If the CTL result is proven there is an escape waveform to look at.

An interesting thing is the addition of constraints can expose type-A deadlocks in a system that does not otherwise have them. A simple example involves reset, where if the design can always be reset, then type-A deadlock is not possible – asserting reset is the escape option.

Adding constraints that can expose a bug may be a new concept for verification engineers used to writing properties. Typically for SVA (or any LTL-based language), constraints are added to remove a CEX that has illegal stimulus. For deadlock verification, correctly specifying constraints is a key to finding system deadlocks.

In the case where an escape CEX is show you must analyze it. If an escape route is invalid (e.g. the reset example mentioned above which is valid but not something you want to have to do in the design), that route can be constrained away such that the reset is no longer valid as an escape mechanism. If the loop CEX is invalid then the loop can be constrained away (e.g. A 'req' happens, then an 'ack' eventually happens). This type of assumption is called a ***fairness constraint***.

There is a prescribed process you can followed with this method using both Case A and Case B scenarios:

1) If Case A type CEX, the design is deadlocked, there is a bug to fix.
2) If Case A type proof, it still may be possible to have a deadlock.
   Note: Case A is a bug hunting technique, a CEX means you have a bug, a proof isn't a guarantee there isn't a bug. Further constraints are needed.
3) If Case A is proven, a Case B CEX will show an escape route from the loop as shown below in Figure 3:

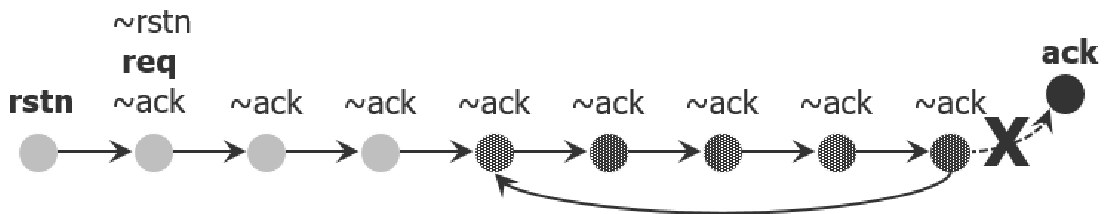

Figure 3: Case B CEX showing an escape route where the ack happens, constrain these away.

4) Add constraints to remove "invalid" escape routes. (e.g. constraining the reset to its inactive state or disabling test/scan).
5) Continue to iterate on adding constraints until there are no more Case B CEX in which case your design is deadlock-free (as long as your constraints are correct!) OR there is a Case A CEX in which case there is a deadlock condition and you have a new bug to fix.
6) When all the bugs are fixed you will have a proof on your original liveness property meaning no deadlock.

With this process typically the number of illegal escape paths is small and deadlock scenarios are typically quickly found when they exist. A major benefit of this flow is that with this type of checking fairness constraints aren't needed to find deadlock. Any proof you have by adding assumptions/constraints is dependent on the validity of the assumptions/constraints. These should be verified as you would do with any formal analysis.

IV. A Simple FSM Example

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

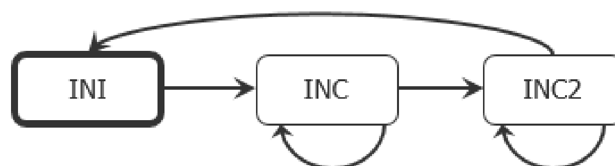

Figure 4: A Simple FSM

The accumulator operates based on the states of the FSM as shown below:
FSM = INI => cntr retains its value
FSM = INC => cntr increments by the value of the input
FSM = INC2 => cntr 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 cntr output:
FSM = INI: Move to state INC
FSM = INC: Remains in this state unless in the input = 2, then moves to state INC2
FSM = INC2: Remains in this state until the value of the counter = 0, then moves to state INI

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

```
module accum (input logic clk, rstn, [1:0] din, output logic [3:0] cntr);

   typedef enum logic [1:0] { INI, INC, INC2 } State;
   State c_state,n_state;

   always_ff @(posedge clk or negedge rstn)
     if (~rstn) cntr <= 4'h0;
     else cntr <= (c_state == INI)   ? cntr :
                  (c_state == INC)   ? cntr + din :
                /* c_state == INC2 */ cntr + 2*din ;

   always_ff @(posedge clk or negedge rstn)
     if (~rstn) c_state <= INI;
     else       c_state <= n_state;

   always @*
       case (c_state)
         INI     : n_state <= INC;
         INC     : n_state <= (din  == 2'd2 ) ? INC2 : INC;
         INC2    : n_state <= (cntr == 4'h0 ) ? INI  : INC2;
         default: n_state <= INI;
       endcase

endmodule
```

Figure 5: Accumulator RTL code to demonstrate deadlock cases for Case A and Case B.

The properties to check if the FSM states can become deadlocked are shown below in Figure 6:

```
bind accum chk_deadlock bind_accum (.*);

module chk_deadlock (
input clk,
input rstn,
input [1:0] din,
input [3:0] cntr
);

typedef enum logic [1:0] { INI, INC, INC2 } State;
State fsm;
assign fsm = accum.c_state;

a_deadlock_ini:  assert property (@(posedge clk) s_eventually(fsm != INI)  );
a_deadlock_inc:  assert property (@(posedge clk) s_eventually(fsm != INC)  );
a_deadlock_inc2: assert property (@(posedge clk) s_eventually(fsm != INC2) );

endmodule
```

Figure 6: Deadlock properties for FSM in accumulator design

In the above example, the results from the formal run would show 1 Proof and 2 Firings or Counter Examples (CEX). Let's take each case in order with the assumption that toggling reset is off the table:

1) INI Deadlock Check Proven: There is no deadlock in this state
2) INC Deadlock Check Fired: Case B deadlock, there is an escape path
3) INC2 Deadlock Check Fired: Case A deadlock, truly deadlocked!

For item one, if the input doesn't have a value of 2, the FSM is deadlocked in the INC state. An escape waveform is shown automatically where in the input is 2 and hence the FSM transitions to the next state, INC2. If the Case B CEX are invalid, then adding constraints could show a true deadlock, or another Escape waveform. A fairness constraint could be added and in this case this check would then be proven. An escapable waveform is shown below in Figure 7 for the 2nd case showing how to escape the INC state:

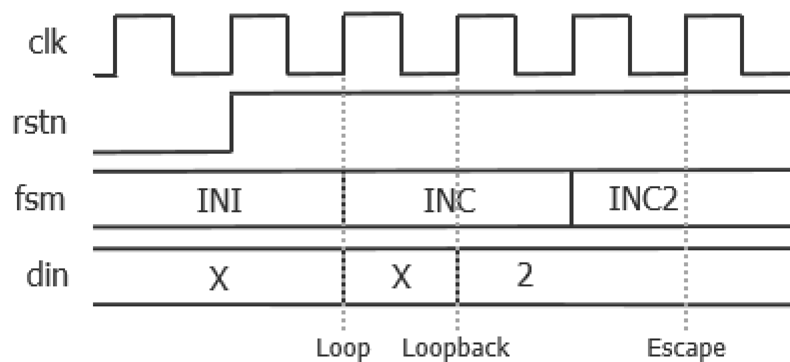

Figure 7: Waveform showing how to escape from the FSM=INC deadlock condition (Case B Example)

The more interesting case is the true deadlock that happens for the INC2 FSM state. This state isn't deadlocked assuming the inputs are behaving in a certain way. However, for a specific sequence of events, 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 INC2:

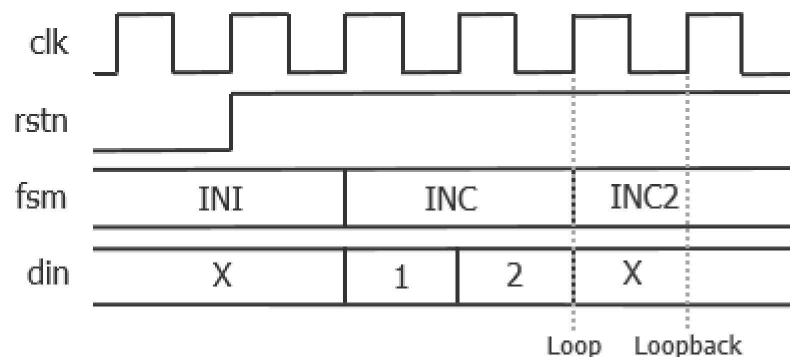

Figure 8: Waveform showing true deadlock for FSM=INC2 (Case A Example)

The deadlock condition in this design occurs when the counter is an odd number when the FSM enters the INC2 state. When the FSM is in state INC2, the counter will always increment with an even number. Since the counter is starting from an odd state, and always incrementing with an even number, the counter will never be 0. Thus the FSM will never exit the INC2 state since that only occurs when the counter hits 0. How does this apply in the real world?

V. Some Deadlock Property Examples

There are various design types 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 scenarios, bus interfaces, and arbitration logic. The idea is to check that some state will either happen like the presence of an 'ack' signal or not happen like the example above where the FSM doesn't stay in a specified state. Generally properties will be described in 2 ways:

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

You don't need to specify that the 'req' happens to check if it is possible for the 'ack' to show up, since formal will put the design into the state such that the check can happen. Also, in this type of analysis there is no need to add a fairness constraint on the 'ack' signal. If you look at the properties from the accumulator design above you will notice the properties 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. Implication can be used to specify some qualifying condition, for example:

```
mode == `WRITE && in_flight && req |=> s_eventually(ack)
```

So, to verify whether an 'ack' will ever follow a request 'req', you would apply the following assertion to check for deadlock. Another example showing a clear for a 'req' is also shown:

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

q_req0_clr0_deadlock: assert property (@(posedge clk) disable iff (!rstn)
                                flow_req[0] |-> s_eventually(flow_req_clr[0]));
```

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

```
a_arb5_deadlock: assert property (@(posedge clk) s_eventually(~grant[5]));
```

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

For verifying deadlock on bus interfaces, for example APB4, the ready signal can be used to check if the interface stays deadlocked in a transaction. Here is an example deadlock property:

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

Normally for this type of check there wouldn't be preconditions added. However, if there is an inconclusive the check could be split up into a read and write checks to further constrain the formal analysis:

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

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

In a similar fashion, to verify that a bus interface like the AXI4 slave address/data channels, will have a ready and valid signals and not stay deadlocked, the final property could be formally analyzed against the design:

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

You could write similar properties for the 'arready', 'wready', 'bvalid', and 'rvalid' signals.

Note: For bus interface deadlock checks the user can specify assertions either on the master side or the slave side.

For cache type designs there are a number of properties that can be written. For example a snoop req/ack type scenario:

```
a_snoop_req_ack_deadlock: assert property (@(posedge clk)
               l1_l2snoopreq && (l1_l2snoopid == 2'd2) |->
               s_eventually(l2_l1snoopack && (l2_l1snoopid == 2'd2)) );
```

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 safety properties. It is beyond the scope of this paper to go into techniques used by formal technicians to resolve inconclusives to get a result from formal analysis. A technique that can be used 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_rxdat_last_deadlock: assert property (@(posedge clk)
          rxdat_act |-> s_eventually(rxdat_act && rxdat_last) );

a_rxdat_rdy_deadlock: assert property (@(posedge clk)
         rxdat_act |-> s_eventually(rxdat_rdy) );
```

Finally, note this important caveat: when applying deadlock verification, it is recommended that the user verify the design for correct functionality before applying these deadlock techniques. Normally a user would run formal using safety assertions as well as verification IP for checking bus interfaces to find any functional bugs in the design. For example you would want your AXI4 interfaces functioning correctly before checking for deadlock scenarios, your FSM's behaving correctly, your arbiters able to arbitrate on each request/grant correctly, req/ack 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. Ideally thee formal analysis will be focused on deadlock scenarios instead of normal functional bugs. When the design is functioning correctly, it is more likely that complex deadlock bugs can be found.

VI. Conclusion

The risk of a design going into deadlock cannot be effectively verified 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 even for formal experts. Fortunately, LTL and CTL property semantics have been embedded in Mentor's PropCheck formal property checking tools to simplify the detection of design deadlock and uncover hidden escape scenarios. This innovative solution allows design and verification engineers to write traditional SVA liveness properties and quickly determine if a deadlock scenario is a Case A or Case B type as described above 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.

[5] DAC 2020, 4.1 – "*Easy Deadlock Verification and Debug with Advanced Formal Verification*", L. Arditi, et.al.