Compositional Reasoning Gotchas in Practice

Chirag Agarwal*, Paul Hylander†, Yogesh Mahajan‡, Jonathan Michelson‡ and Vigyan Singhal§
*Oski Technology, Gurgaon, India
†Ikanos Communications, Fremont, California, USA
‡NVIDIA, Santa Clara, California, USA
§Oski Technology, Mountain View, California, USA

***Abstract*—Model checking has become a formal sign-off requirement in the verification plans of many hardware designs. For design sizes encountered in practice, compositional assume-guarantee reasoning is often necessary to achieve satisfactory results. However, many pitfalls exist that can create unsound or unexpected results for users of commercial model checking tools. Users need to watch out for circularity in properties, for dead-ends getting trimmed by tools, as well as understand the differences in proof composition for liveness and safety properties. We present many real design examples to illustrate these points, as well as describe our experiences with compositional reasoning in practice.**

I. Introduction

Compositional proofs, while highly desirable, are sometimes tricky to apply correctly in practice. Compositional reasoning is probably the most widely used form of assume-guarantee reasoning. Assume-guarantee reasoning does not necessarily cut out the guaranteeing logic, e.g. when establishing inductive invariants or helper lemmas that simplify the proofs for some properties. Compositional reasoning focuses on cutting out the guaranteeing logic when assuming the property which has been guaranteed, thereby reducing complexity by analyzing smaller chunks of logic. Cutting out logic is often implemented by running model checkers on sub-modules or by black boxing some sub-modules.

Compositional reasoning is naturally suited to hardware designs, which are parallel compositions of thousands or millions of processes. Adoption of compositional reasoning is also driven by the existence of interfaces, which are natural boundaries for contracts between the designers. The contracts may or may not always be explicit, but there's a good chance that a well-designed hardware interface obeys some relatively simple contracts, as would be consistent with following good design principles like encapsulation. The designed interfaces and protocols are intended to guarantee high-level properties that are targets of verification for us. Using compositional reasoning and relying on interfaces, we can verify higher-level system properties, such as absence of system-level deadlocks.

Given the size of industrial designs, the system-level verification problem is impractical to solve in a formal verification setup using the entire chip system as the design-under-test (DUT). Rather, one aims to find related proof obligations on the logic of smaller sub-units (typically coded by one RTL designer), which when taken together are sufficient to imply that the original system properties hold. This is the most practical way that model checkers are able to verify system properties to the level of confidence desired. As a side benefit, using assume-guarantee for properties on interfaces between neighboring units, both of which have model checking setups, can yield insight about design invariants and ultimately help formalize the inter-designer contracts that may not have been explicitly or precisely specified earlier.

However, as we discuss in this paper, it is important to figure out the process to get compositional reasoning right. There are plenty of pitfalls in this activity, and we would like to publicize some of the gotchas.

Our discussion in this paper is based only on our practical experience in a setting that is limited to applying commercial model checking tools on synthesizable RTL designs while verifying properties written in the popular SystemVerilog Assertion (SVA) language [1].

In Section II, we describe some previous work on compositional reasoning, and some challenges in using that in our setting. In Section III, we describe how we apply compositional reasoning. We need to watch out for false positives when tools trim dead-ends silently (Section IV). In Section V, we caution about changes in the SVA liveness syntax and go on to detail an example of a missed deadlock bug when compositional reasoning is applied for liveness properties without appropriate care. We conclude in Section VI by pointing out some steps end-users can take to avoid such pitfalls.

II. Prior Results and Discussion

When applying compositional assume-guarantee reasoning, it is important to be able to tell what may be safely inferred about properties at the system level based on the results seen at the unit level. To begin with, one may ask if it is sound to conclude that some high-level property holds at the system level if all unit level obligations are proven. This is a reasonable question since one can encounter circularity in the assume-guarantee argument, and circularity in the argument could lead to unsoundness. For example, given two neighboring modules A and B, property P_A might be verified on A assuming P_B holds, and then P_B might be verified on B assuming P_A holds, which is a form of circular argument. To help address the issue of potential unsoundness in the compositional reasoning arguments, there are many theoretical results which characterize sound applications of compositional reasoning. The work by McMillan [2] is widely known. A good overview of compositional reasoning with a focus on completeness and soundness is presented by Namjoshi and

$$\begin{tikzcd}
p_2 \arrow[r] \arrow[dr] & p_4 \arrow[r] & p_3 \\
 & p_1 \arrow[ur] &
\end{tikzcd}$$

$\mathcal{O}$

$$(p_2 \wedge p_4) \rhd (p_2 \Rightarrow p_1)$$
$$(p_1 \wedge p_3) \rhd p_2$$
$$\texttt{true} \rhd ((p_1 \wedge p_4) \Rightarrow p_3)$$
$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4) \rhd (p_2 \Rightarrow p_4)$$

Fig. 1. Example illustrating some concepts involved in applying McMillan's compositional rule for 4 properties p_1, p_2, p_3, p_4. The graph $\mathcal{O}$ is used to order the properties. On the right are shown four target proof obligations that are consistent with applying this rule.

Trefler [3]. An alternative scheme for compositional reasoning is presented in follow up work by Amla *et al.* [4].

A. Terminology and Notation

We use SVA to write properties. SVA extends Linear Temporal Logic (LTL) [5] with operations which increase expressiveness[1] and succinctness. An assertion written as **assert P** indicates that property **P** is expected to be `true` starting anywhere along the trace. If property **P** corresponds to temporal logic formula p, **assert P** corresponds to the temporal logic formula $G(p)$. Similarly, an assumption written as **assume P** requires that $G(p)$ does not fail.

When discussing the soundness results, it is useful to know the distinction between *safety* and *liveness* properties [6]. A safety property is one whose failures can be witnessed by finite traces. Once a safety property is witnessed to fail, no further extension of the trace can make the property hold. A liveness property is one whose failures cannot be witnessed by finite traces. For a liveness property, every finite trace can be extended such that the property does not fail, i.e. every failure trace is of infinite length. Every property can be rewritten as a conjunction of a safety property and a liveness property. In this paper, we will assume that every property is written as either a safety or liveness property.

The suggestive notation $q \rhd p$ (read as "q constrains p") is used below for consecution claims. $q \rhd p$ means that it is never the case that p is `false` in cycle n and q is `true` in all prior cycles. It is equivalent to $\neg(qU\neg p)$.

B. McMillan's circular compositional reasoning rule

McMillan's compositional reasoning result provides a sufficient condition for concluding that $G(\bigwedge p_i)$ holds for the design given that some local proof obligations are met. Each proof obligation takes the form of a consecution claim, i.e. something is claimed about cycle n of a trace if a related claim applies during cycles $0 \ldots (n-1)$ of the trace. We will consider only the case with a finite number of properties p_i.

A key part of McMillan's result involves partially ordering the properties, or equivalently, viewing the properties as nodes of some acyclic directed graph $\mathcal{O}$.

The main result is that one can conclude that $\mathcal{S} \models G(\bigwedge p_i)$ if we establish for all i that $\mathcal{A}_i \models \{\Delta_i \rhd ((\bigwedge_{p \in T_i} p) \Rightarrow p_i)\}$, where

[1]Unlike LTL, SVA has the expressive power of ω-regular languages.

i. $\mathcal{S}$ is the original design with its environment constraints
ii. $\mathcal{A}_i$ is a valid abstraction of $\mathcal{S}$ (typically obtained by retaining the guaranteeing logic for p_i and cutting out unrelated logic, e.g. via black-boxing some modules)
iii. $T_i \subset \{p_1, p_2, \ldots, p_k\}$ such that $p_j \in T_i$ implies that there is a (nontrivial) path from p_j to p_i in the acyclic graph $\mathcal{O}$
iv. Δ_i is $(\bigwedge_{p \in D_i} p)$ with $D_i \subseteq \{p_1, p_2, \ldots, p_k\}$

(An illustrative example showing an application of the rule is in Fig. 1 for the case of 4 properties.)

In this result, the key part is the definition of T_i using the ordering implied by $\mathcal{O}$. This allows one to construct an argument that $\bigwedge p_i$ is not `false` at cycle n assuming $\bigwedge \Delta_i$ is not `false` at all prior cycles. This then allows one to conclude that Δ_i is not `false` at cycle n and enables the inductive step for the next cycle.

The result applies irrespective of whether the p_i's are safety or liveness properties. Since liveness properties do not fail on finite traces, and all failures of safety properties are witnessed by finite traces, the inductive argument may be somewhat surprising! The potential for confusion arises because saying "p_i is not `false` at cycle n" is not the same as saying "p_i does not fail at cycle n", especially when p_i is not a combinational property. Each property p_i is itself a path formula which can be evaluated as `true` or `false` for the path beginning in cycle n. A (safety) property p_i fails at cycle n if cycles m through n of the trace form a bad prefix for p_i for some $m \leq n$.

For the case where all the p_i are safety properties, the induction argument *can* in fact be of the form that $\bigwedge p_i$ does not fail at cycle n assuming $\bigwedge \Delta_i$ does not fail at any prior cycle. Then Δ_i does not fail at cycle n and enables the next inductive step. In this case, the induction is over the length of finite traces and we seek the shortest trace which witnesses a failure of the safety properties involved.

C. Challenges applying McMillan's result

Implementing McMillan's compositional assume-guarantee result as given above is not simple in practice:

i. For a large number of properties, providing and maintaining a valid and effective ordering of the properties can get unwieldy and burdensome for the end user.
ii. Writing the compositional proof obligations using SVA can get complicated. For example, writing $\neg(qU\neg p)$ in SVA naively as "**assert property** `(not (q s_until (not p)));`" actually expresses $G\neg(qU\neg p)$ which evaluates the path formula $\neg(qU\neg p)$ at each cycle of the trace. We wish to evaluate $\neg(qU\neg p)$ only at the first cycle.

In addition to the above issues which are specific to applying McMillan's result, there is also the broader question of how a particular compositional reasoning result applies when restricted to finite traces. This is important because we frequently reason about finite traces in practice. Which semantics for finite traces are compatible with the result of a given compositional reasoning theorem? (For example, the safety-only variant of the argument above which uses the notion of "p_i does not fail in cycle k" has a dependency on the semantics

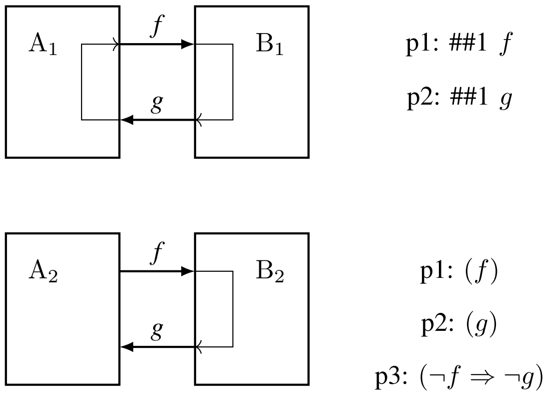

Fig. 2. Zero-delay loops

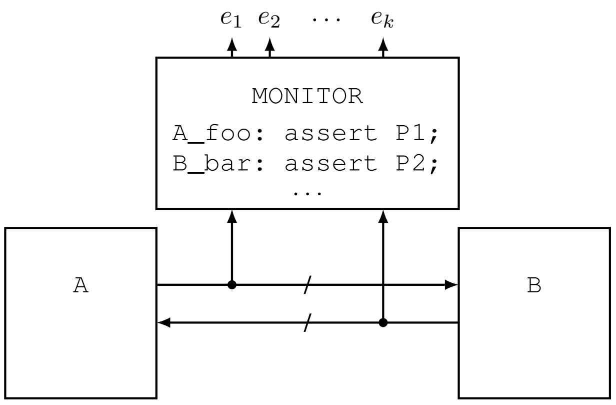

Fig. 3. Structure used for compositional reasoning about safety properties

used for finite traces. The user needs to confirm that the semantics SVA uses for finite words do not break the inductive argument. Further, certain related tool settings/behaviors like trimming dead-ends can break the inductive argument and lead to unsound results when reasoning about finite traces.)

D. Another Compositional Reasoning Approach

Since McMillan's rule can be challenging to apply, a simpler compositional reasoning rule as described below is often used.

Model checking setups are created to check properties on abstractions $\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_m$. Each property p_i is associated with the abstraction it is intended to be checked on. Let M_k be the set of all properties intended to be proven on $\mathcal{A}_k$.

In order to claim that $\mathcal{S} \models G(\bigwedge p_i)$, establish for all k that $\mathcal{A}_k \models G(\bigwedge_{p \notin M_k} p) \Rightarrow G(\bigwedge_{p \in M_k} p)$, where

i. $\mathcal{S}$ is the original design with its environment constraints
ii. $\mathcal{A}_k$ is a valid abstraction of $\mathcal{S}$ containing the guaranteeing logic for all $p \in M_k$

Unlike in McMillan's rule, there is no longer any requirement to explicitly order the properties. Further, SVA can express the proof obligations very simply. In the setup for $\mathcal{A}_i$, use **assume P** for all $p \notin M_i$ and **assert P** for all $p \in M_i$.

Unfortunately, this argument is not sound in general, as summarized in detail in [4]. It is unsound for liveness properties. It can be used only with safety properties, but that too has a few corner cases where it can yield unsound results.

The tradeoff is that the user now needs to be aware of the assumptions under which this compositional reasoning claim is valid and to have a way of knowing when these extra assumptions are getting violated, in addition to the problem of not accidentally using tool settings that interfere with the induction argument.

Since the compositional reasoning argument for safety properties involves an inductive argument on the length of the shortest trace which violates one of the safety properties, the potential for unsoundness lies in either the base case or the inductive step being spuriously claimed to hold.

One way the inductive step can spuriously be claimed to hold occurs when there are zero-delay loops in the logic involved. Fig. 2 pictorially depicts two ways a zero-delay loop can occur in the logic.

In the first case shown in the upper half of Fig. 2, there is a combinational loop going through modules A_1 and B_1, and we are skipping the check in the cycle after reset. All tools complain about such combinational loops, and so we need not worry about this pitfall.

In the second case shown in the lower half of Fig. 2, property *p1* will get proven on A_2 after assuming *p2* and *p3*, irrespective of the driving logic of f (assuming the base case holds at reset), and properties *p2* and *p3* will get proven on B_2 assuming *p1*. This situation is due to a zero-delay loop in the logic involving f getting reflected as g by B_2 combined with the structure of the properties. In this case, unfortunately, tools do not complain about the combinational zero-cycle dependency, and the user needs to be careful that the properties do not cause such a loop.

Further examples illustrating unsoundness are presented in Section IV (safety case) and Section V-B (liveness case). Section III briefly describes the methodology used in these unsoundness examples.

III. Our Methodology

The structure we use for compositional assume-guarantee reasoning is illustrated in Fig. 3 for interconnected modules A and B. It uses an SVA bind to hook a monitor module to the interface between the modules. The model checking setup for A black-boxes B and vice versa. Properties are written in the monitor module using a naming convention where the name of the property indicates which module is expected to contain the guaranteeing logic for the property. When running the setup with A as the DUT, the properties in the monitor with A expected to be guaranteeing logic are used as SVA asserts, while the properties expected to be guaranteed by B are used as SVA assumes. For example, the property named A_foo is used as an assertion for A and an assumption for B. If the properties asserted on A and B do not fail in their respective setups, it is claimed (possibly unsoundly) that the properties hold for the composed system up to the minimum bounded proof depth achieved. This structure implements the approach described in Section II-D. It is unsound to use this structure for composing liveness properties (refer to Section V-B).

Note that for reset, we (and tools) assume that the hardware logic resets to a state consistent with 3-valued simulation. Our SVA assumptions are disabled during the reset analysis phase.

IV. Deadends and Impact of Trimming Deadends

Writing constraints[2] is often a challenging and time-consuming endeavor. Over time, users of formal verification tools have developed their own guidelines to converge on a "good" set of constraints. Some different guidelines are:

- Begin with the environment as under-constrained as possible, so you avoid missing bugs by accidentally over-constraining the setup.
- Begin with an over-constrained setup, to minimize false failures and avoid inefficient use of designer debug time.
- Avoid writing constraints on the outputs or on internal signals of the DUT.
- If possible, write constraints as implications with the constrained input appearing in the consequent of the implications.
- Avoid dead-ends.

A. Intentional dead-ends

Dead-ends are artifacts of using constraints. In any RTL design, without any constraints, any sequence of input values is permitted (and the design will produce some output, no matter what the input sequence is). In the presence of constraints, however, we can have permissible finite sequences of input values that cannot be extended any further if every choice of the next input value violates at least one constraint. Dead-ends are recursively defined as states from which either no transition is possible (0-cycle dead-ends), or the only transitions are to states that are dead-ends (multi-cycle dead-ends).

The SystemVerilog standard [7] does not mandate what the tools should do with dead-ends, and unfortunately, different commercial tools treat dead-ends differently – some tools never trim dead-ends (i.e. a finite trace is considered a legal counter-example to a safety property even if it ends in a dead-end) unless explicitly instructed by the user, whereas other tools trim dead-ends, and sometimes they trim 0-cycle and 1-cycle dead-ends, but not other dead-ends!

The above-mentioned guideline of writing constraints carefully to avoid dead-ends, is religious, and different opinions exist. Sometimes the process of avoiding dead-ends can make the code less intuitive or less readable. Consider a design with one input *a*, and where *a* should be constrained such that it is never asserted in three consecutive cycles, nor deasserted in three consecutive cycles. One popular method of implementing constraint models is through state machines. The state machine in Fig. 4 can be used to implement this constraint via the following code:

[2]In the following sections, we will use the word *constraints* to refer to assumptions, because the use of *constraints* as well as *under-constraints* and *over-constraints* is more popular in commercial tools.

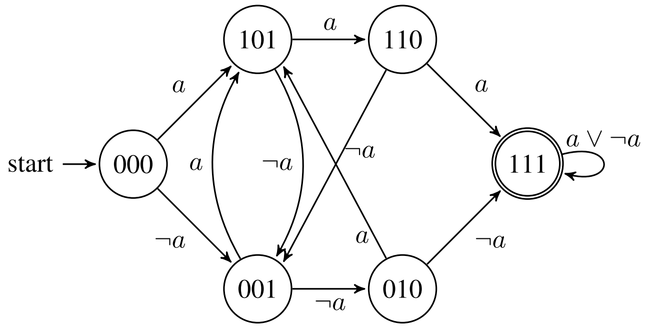

Fig. 4. Constraint implemented using a state machine

```
no_000_111_c: assume property(
  @(posedge clk) disable iff(!rstn)
      sm != 3'b111
);
```

The state 111, the error state, is a dead-end. Using the code above, the user intends for the tool to remove any input sequences that will cause the state machine to go to this error state. Consider a 5-cycle long input sequence $0{\cdot}1{\cdot}0{\cdot}0{\cdot}0$, which violates the desired constraint that *a* should not be deasserted for 3 consecutive cycles, causing the state machine to arrive in state 111 in the 6-th cycle. If some design assertion fails on the 5-th cycle on this input sequence, the questions is whether that failure is a legal counter-example or not. Likely, the author of this constraint code considers that sequence illegal, and would not be happy with the false failure. We think that might be the reason why some tools trim dead-ends, perhaps in response to such seemingly reasonable expectations. In fact, using the same state machine, coding constraints to avoid any dead-ends in the first place would require code like the following:

```
no_000_c: assume property(
  @(posedge clk) disable iff(!rstn)
      (sm == 3'b010) |-> a
);
no_111_c: assume property(
  @(posedge clk) disable iff(!rstn)
      (sm == 3'b110) |-> (!a)
);
```

This is more verbose than the previous code, and one can imagine it being even more so, if there were more arcs to the error state.

B. Compositional reasoning with dead-ends

Even though it might seem reasonable for tools to trim dead-ends, we show a serious problem that can happen during compositional reasoning due to this.

Consider modules A and B shown in Fig 5. B receives packets sent by A, buffers them in a FIFO structure which is 8 entries deep, and then sends the packets downstream. A is responsible for making sure that it does not send more packets than B can hold, while B is responsible for notifying A when packets are being drained from the FIFO. The interface signals depicted in the figure show a simplified view of the flow control used in the actual design – A sets signal *push*

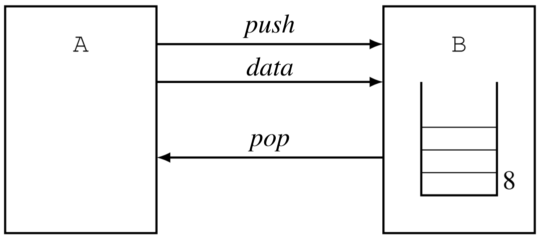

Fig. 5. A sends packets to B. B buffers the packets in a FIFO of size 8.

when valid data is sent and B uses signal *pop* to indicate when an entry has been dequeued from the FIFO.

The checks that the FIFO does not overflow or underflow are implemented using a counter *ctr* in the interface monitor as follows:

```
logic [3:0] ctr;
always @(posedge clk or negedge rstn)
begin
  if (!rstn)
    ctr <= 0;
  else
    ctr <= ctr + push - pop;
end

A_no_overflow: assert property(
@(posedge clk) disable iff(!rstn)
  ctr + push <= 8
);

B_no_underflow: assert property(
@(posedge clk) disable iff(!rstn)
  pop <= ctr
);
```

Since *ctr* is 4-bit wide, an underflow makes it wrap around to 15. Suppose the design has a bug where B sends *pop* when *ctr* is 0, but this bug can only happen when *push* is 0. When this failure of *B_no_underflow* happens, *ctr* will have the value 15 in the next cycle and that will cause *A_no_overflow* to fail. Since *A_no_overflow* is used as an assumption when checking *B_no_underflow* on B, the failure of *B_no_underflow* is witnessed only by 0-cycle dead-end traces.

If the user enables the tool setting to hide failure traces that cannot be extended by at least one more cycle, or if the tool trims dead-ends by default, then the tool will trim all failures of *B_no_underflow* and report it as proven.

We recommend implementing constraints so dead-ends are avoided altogether, even if that makes the code more verbose, as in Section IV-A. For the current example, one way to avoid the dead-ends is to rewrite the *A_no_underflow* property as:

```
A_no_overflow: assert property(
@(posedge clk) disable iff(!rstn)
  (ctr == 8) |-> !push
);
```

The above approach to removing unintentional dead-ends is reactive. We would instead like the tools to support a check that the formal testbench permits no dead-ends, and if the check fails, to produce a finite witness ending in a dead-end.

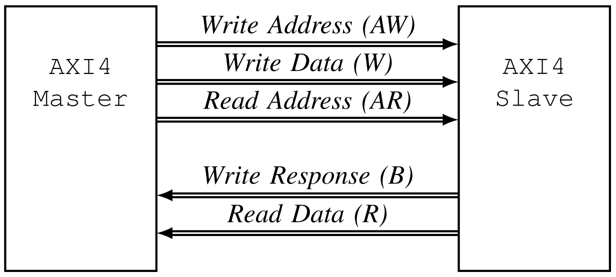

Fig. 6. ARM AMBA AXI4 interface

V. Compositional Proofs of Forward Progress

We describe a system we have seen where a deadlock may be missed in the process of compositional proofs, if the properties are expressed as liveness properties.

A. Liveness syntax and semantics in SystemVerilog 2009

Before going on to describe the missed deadlock, we want to alert the reader that the SystemVerilog semantics adopted in 2009 (IEEE Standard 1800-2009 [7]) significantly change the meaning of property expressions like the one below:

```
a |-> ##[0:$] b
```

Before 2009, this syntax was used to express the property that if *a* is true, eventually *b* must be true [1]. However, with the change in the standard, the property expression written above is equivalent to `true`, even if *a* and *b* are primary inputs with no constraints on them! This was a surprise to the authors, and given that tools do not typically warn the users that such property expressions are very likely not doing what the user intended, we consider this very dangerous.

In contrast, the tools do error out on a property expression like the one below which is deemed illegal:

```
a |-> eventually b
```

The safer way to express liveness in SystemVerilog 2009 is using syntax like the following:

```
a |-> s_eventually b
```

B. AXI4 deadlock missed while composing liveness properties

This design[3], a simplified version of a real-system design, is based on the popular ARM AMBA AXI4 on-chip interface standard [8]. The AXI4 standard connects a master to a slave via five asynchronous channels (Fig. 6): Write Address (AW), Write Data (W), Read Address (AR), Write Response (B) and Read Data (R). The standard assumes a synchronous clock *ACLK*.

The Write Address and Write Data transactions are responded by a single-beat Write Response (B) that indicates whether the write succeeded without errors, or not. Two important signals in this B channel are *BVALID* and *BREADY*. *BVALID* indicates that the slave is sending the write response on this cycle, and *BVALID* stays asserted until the master

[3]The RTL of our simplified design is available at http://www.oskitechnology.com/wp-content/uploads/2015/09/fmcad15.tar.gz

acknowledges the response with *BREADY*. Similarly, the Read Data (R) channel is used to send the read response, which includes the read data. Besides the read data itself, three important signals in the R channel are *RVALID*, *RLAST* and *RREADY*. *RVALID* is asserted on each cycle the slave is sending the read data, and since the read could be a burst read, *RLAST* is used to indicate whether the current beat of data is the last beat. Like with the B channel, the ready signal *RREADY* is used by the master to indicate to the slave that it has accepted the current beat – until that happens, the slave is required to hold it values of *RVALID*, *RLAST* and read data.

To describe the deadlock situation, it is convenient to assume that each read is either 1 or 2 beats, no longer. We will only need to look at the B and R channels. Two AXI4 properties will participate in this deadlock, one master property (*M*) and one slave property (*S*):

1) master property *M*: once the master receives *BVALID*, eventually it must assert *BREADY*
2) slave property *S*: once the slave sends *RVALID* with *RLAST* deasserted (i.e. not for the last beat) and it is accepted by the master which asserts *RREADY*, eventually the slave must send *RVALID* with *RLAST* asserted

These two properties can be implemented in SystemVerilog 2009 as the following liveness properties:

```
property master_liveness_bready;
  @(posedge aclk) disable iff (!aresetn)
    (bvalid && !bready) |->
      s_eventually bready;
endproperty

property slave_liveness_rlast;
  @(posedge aclk) disable iff (!aresetn)
    (rvalid && !rlast && rready) |->
      s_eventually (rvalid && rlast);
endproperty
```

The AXI4 protocol allows the B and R channels to be completely decoupled from each other. However, to optimize resources or area, some master or slave may choose to share a FIFO for both the B and R responses. We will call such a device a serializing device.

The deadlock happened because a serializing master was connected to a serializing agent. This deadlock scenario is depicted in Fig. 7. In the simplified version of the deadlock, the serializing master has a 2-deep FIFO that stores the B and R responses. For the R responses, the master needs to receive the entire read response (whether it is 1 beat or 2 beats), before it dequeues the read response from the queue. For design-specific reasons, the master processes the requests in order – so if a B response arrives later than a previous R response, the prior R response must be processed before the B response is accepted by asserting *BREADY*. Similarly, the serializing slave also has a 2-deep FIFO that stores the B and R responses that are queued up to be sent to the master. For the deadlock to happen, we have three transactions: a Read Response R1 composed of two beats R1.F and R1.L, and two Write Responses B1 and B2. The slave decides to send R1.F, followed by B1, followed

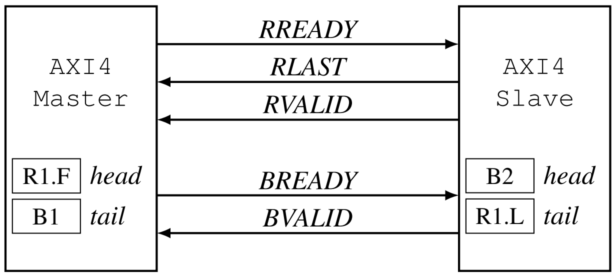

Fig. 7. AXI4 system deadlock

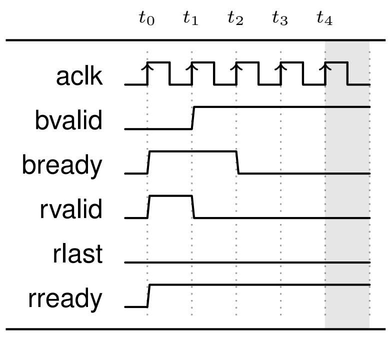

Fig. 8. Deadlock failure in the composed design (last 1 cycle loops forever)

by B2, followed by R1.L. This results in a deadlock shown in Fig. 7: the master does not assert *BREADY* for B2, because it is waiting for R1.L. The slave will not send R1.L, until B2 is dequeued first, causing the deadlock.

In fact, if the model checking setup has the DUT as the entire system containing both the master and the slave RTL, each of the two liveness properties fails, with infinite-length counter-examples showing the deadlock (Fig. 8 shows one of these two failures). The reader may observe that the deadlock can be avoided by forcing either the master or the slave to be non-serializing, and in that sense, it is arguable if the deadlock is due to a bug in the master or in the slave!

However, the situation becomes interesting when we use compositional reasoning. In the real system, the master and slave modules were large enough (and designed by different RTL designers) that it was important to verify the modules separately. We wanted to prove the property *M* on the master, and the property *S* on the slave. Since each module is serializing, and depends on fairness constraints from the other, it seems natural to want to prove *M* on the master while assuming *S*; and conversely, to prove *S* on the slave while assuming *M*. The methodology described in Section III was used to carry out this compositional argument. An expert reader may realize at this point that each of these liveness checks may now actually pass. In fact, that is exactly what happens! A naive user might then incorrectly conclude that *M* and *S* are true on the composed system and miss the deadlock bug.

When using the methodology of Section III, the proof decomposition attempted is to prove $G(M) \Rightarrow G(S)$ on the slave and $G(S) \Rightarrow G(M)$ on the master. Suppose it happens that whenever S is `false` for the slave, property M must also necessarily be `false` somewhere further along the same trace. Then if we assume $G(M)$ when proving $G(S)$, we will no longer see any failures for $G(S)$.

Suppose we had instead tried to prove $G(M \Rightarrow S)$ on the slave and $G(S \Rightarrow M)$ on the master. This proof decomposition would be equivalent to checking `true` $\triangleright (M \Rightarrow S)$ on the slave and `true` $\triangleright (S \Rightarrow M)$ on the master. For this decomposition, we cannot use McMillan's rule to infer that $G(M \wedge S)$ holds for the composed design, because the implied ordering graph would be cyclic $\begin{tikzcd} M \arrow[r, bend left] & S \arrow[l, bend left] \end{tikzcd}$

(Note that `true` $\triangleright p$ is equivalent to $\neg(\texttt{true} U \neg p)$ which can be rewritten as $\neg(F \neg p)$ and then as Gp.)

C. *Using liveness properties safely with McMillan's rule*

To confirm that we cannot miss the deadlock when applying McMillan's circular compositional rule with the liveness properties M and S, we ran the four checks listed below:

i. on the master, check $(M \wedge S) \triangleright S$
ii. on the master, check $(M \wedge S) \triangleright (M \Rightarrow S)$
iii. on the slave, check $(M \wedge S) \triangleright S$
iv. on the slave, check $(M \wedge S) \triangleright (M \Rightarrow S)$

All the above checks fail, implying that all attempts to apply McMillan's circular compositional rule to infer $G(M \wedge S)$ will involve a failing check. Hence, the user will not be led to the incorrect conclusion that there is no deadlock.

For reference, the SVA implementation is shown below:

```
wire mlhs = (bvalid && (!bready));
wire mrhs = bready;
wire slhs = (rvalid && (!rlast) && rready);
wire srhs = (rvalid && rlast);
reg aresetn_d;
always @(posedge aclk) begin
  aresetn_d <= aresetn;
end

property m;
  (mlhs |=> s_eventually mrhs);
endproperty

property s;
  (slhs |=> s_eventually srhs);
endproperty

property T(e);
@(posedge aclk) disable iff (!aresetn)
  e;
endproperty

property F(p);
  !aresetn_d |-> (p);
endproperty

property K(l,r);
  not ((l) s_until (not (r)));
endproperty

chk0_s: // most constrained check
  assert property (
    T(F(K(m and s,s)))
);
```

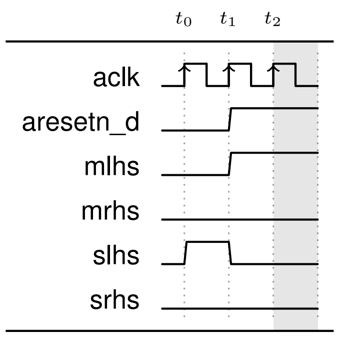

Fig. 9. Failure for chk0_s on the slave (last 1 cycle loops forever). The same trace also shows chk0_m_s failing.

```
chk0_m_s: // most constrained check
  assert property (
    T(F(K(m and s,m implies s)))
);
```

The failure trace for *chk0_s* on the slave is shown in Fig. 9 and shows the property *s* is `false` in the first cycle. Since property *m* is `true` in the first cycle, the same trace is also a failure trace for *chk0_m_s* on the slave.

For sake of completeness, it may be noted that the only checks that do not fail (and get proven) are the following:

i. on the master, check $(M \wedge S) \triangleright M$
ii. on the master, check $(M \wedge S) \triangleright (S \Rightarrow M)$
iii. on the master, check $S \triangleright M$
iv. on the master, check $S \triangleright (S \Rightarrow M)$

D. *Using safety properties to express forward progress*

Next, we explore the use of safety properties for compositional reasoning about forward progress properties. Users are often divided about their preference for using liveness versus safety properties to prove forward progress or absence of deadlocks [9]. Liveness is usually more elegant, although it may require some iterations to identify appropriate fairness constraints. Safety properties can be used by picking a design-specific constant to require the consequent to be satisfied within this constant number of cycles, but it may need some iterations to figure out the value of the constant.

For the example as described previously, the master and slave properties can be written as the following safety variants:

```
property master_safety_bready;
  @(posedge aclk) disable iff (!aresetn)
    (bvalid && (!bready)) |->
      ##[1:`B_TIMEOUT] bready;
endproperty

property slave_safety_rlast;
  @(posedge aclk) disable iff (!aresetn)
    (rvalid && (!rlast) && rready) |->
      ##[1:`R_TIMEOUT] (rvalid && rlast);
endproperty
```

Of course, the two defined constants $B\_TIMEOUT$ and $R\_TIMEOUT$ must be selected to be large enough that

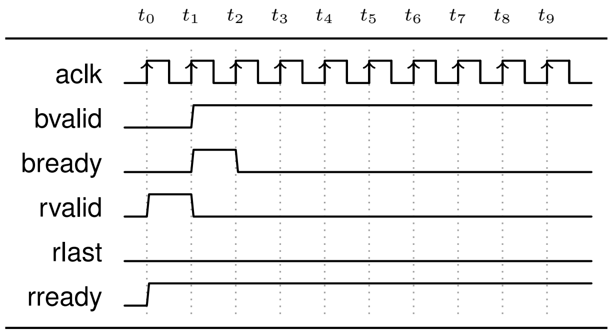

Fig. 10. Slave failure (*B_TIMEOUT=8*; *R_TIMEOUT=8*)

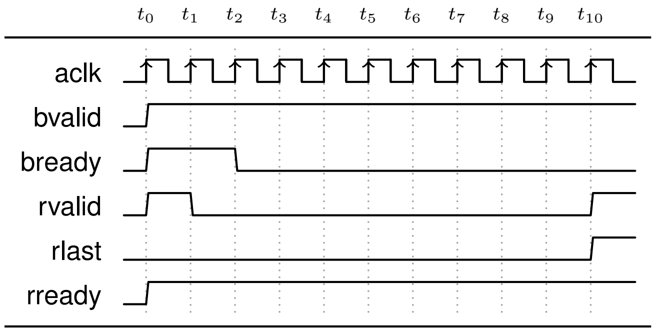
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
aclk
bvalid
bready
rvalid
rlast
rready

Fig. 11. Master failure (*B_TIMEOUT=8*; *R_TIMEOUT=12*)

desirable design behaviors are not flagged as errors. When these properties are checked on a system composed of the master and slave, a value of 8 for each of the two constants shows the deadlock scenario. The counter-example trace is very similar to that in Fig. 8, except that instead of the lasso at the end, the trace is stretched to about 8 extra cycles.

Further, as discussed in Section V-B, for this to work in practice on real-sized designs, we need to prove these two properties separately on the master and slave RTL modules. Choosing constant values of $B\_TIMEOUT = 8$ and $R\_TIMEOUT = 8$, if we assume *slave_safety_rlast*, the master property *master_safety_bready* passes on the master RTL module. However, doing the reverse, while assuming $master\_safety\_bready$, the slave property *slave_safety_rlast* fails with the waveform in Fig. 10.

The root cause of the failure appears to be that the master might be responsible for this since it is not accepting the second *BVALID* in the trace with a corresponding *BREADY*. So, the user is tempted to re-run by increasing *R_TIMEOUT* relative to *B_TIMEOUT*. Choosing constant values of $B\_TIMEOUT = 8$ and $R\_TIMEOUT = 12$, indeed, the slave property *slave_safety_rlast* passes while assuming *master_safety_bready*. However, now the master property *master_safety_bready* fails on the master RTL with the waveform in Fig. 11.

In fact, one can try all possible values of $B\_TIMEOUT$ and $R\_TIMEOUT$, and observe that no matter which values are chosen, either the master RTL or the slave RTL or both show a failure. Table I shows the results for

TABLE I
ASSUME-GUARANTEE RESULTS WITH SAFETY PROPERTIES

B_TIMEOUT	R_TIMEOUT	master result	slave result
8	≤ 9	Pass	Fail
8	10 or 11	Fail	Fail
8	≥ 12	Fail	Pass

$B\_TIMEOUT = 8$, and similar results are seen for other values of $B\_TIMEOUT$. This is a good result, because unlike the liveness situation (Section V-B), a naive user does not have to take on the burden of avoiding the circularity pitfall.

VI. Conclusion

While doing compositional reasoning, users need to be careful about avoiding circularity. They need to be careful that the properties combined with the hardware design do not create zero-delay loops. If tools trim dead-ends, compositional proofs may not work, unless constraints are written in a specific coding style. Liveness properties with compositional reasoning are dangerous unless users have taken care to order properties, or otherwise use McMillan's method accurately.

We hope the examples and our experiences are useful for other users practicing formal verification on hardware designs.

Acknowledgment

The authors would like to thank Prashant Aggarwal and Arun Khurana for experimenting and sharing their results on some design examples we discussed in this paper, and Jin Zhang for her review on an earlier version of the paper. We also thank the reviewers for their useful comments.

References

[1] F. Haque, J. Michelson, and K. Khan, "The art of verification with SystemVerilog assertions," *Verification Central*, 2006.
[2] K. L. McMillan, "Circular compositional reasoning about liveness," in *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME 99), volume 1703 of Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 342–345.
[3] K. S. Namjoshi and R. J. Trefler, "On the completeness of compositional reasoning," in *Computer Aided Verification*. Springer, 2000, pp. 139–153.
[4] N. Amla, E. A. Emerson, K. Namjoshi, and R. Trefler, "Abstract patterns for compositional reasoning," in *In Concurrency Theory (CONCUR), LNCS 2761*. SpringerVerlag, 2003, pp. 423–448.
[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.
[6] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed computing*, vol. 2, no. 3, pp. 117–126, 1987.
[7] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. 1–1285, Dec 2009.
[8] ARM Ltd., "AMBA AXI and ACE protocol specification, issue D," 2011.
[9] B. Krishna, J. Michelson, V. Singhal, and A. Jain, "Liveness vs safety–a practical viewpoint," in *Hardware and Software: Verification and Testing*. Springer, 2012, pp. 80–94.