Liveness Assume-Guarantee Proof Schema: A step towards Liveness Full Proofs

Nitish Sharma, Venkata Nishanth Narisetty
Qualcomm Inc.
{nitshar@qti.qualcomm.com,vnariset@qti.qualcomm.com}

***Abstract*- Formal Verification (FV) has established itself as a critical element of high-quality functional verification, playing an indispensable role in the successful tape-outs of modern hardware designs. While safety properties have received significant advancements in recent EDA tools, liveness properties remain relatively overlooked, even amongst the formal experts. In this work, we introduce a robust mathematical formulation aimed at exploring deep states in the design and uncover liveness issues like starvation, and deadlocks etc. Furthermore, we extend this methodology to mitigate the FV complexity related to liveness, enabling us to achieve exhaustive proofs for liveness properties. We also present a case-study on a simulation signed off design where this approach found multiple issues.**

I. INTRODUCTION

In modern hardware designs, ensuring forward progress and absence of system hangs is paramount for a successful product release. With the increasing complexity of features, and shrinking market deadlines, the stress on dynamic simulation has grown, leading to last-minute panics and even bug escapes. Traditionally, the Design Verification (DV) team has relied on End-of-Simulation (EoS) checks, primarily focused on identifying hang issues in the system. However, dynamic stimulus for forward progress issues, which necessitate the interaction of multiple agents in specific event sequences, has been found to be lacking in practice. Even the Formal Verification (FV) struggles to provide any meaningful assistance due to exponential computational complexity within an end-to-end verification environment.

To address hardware system behavior over an infinite number of cycles, System Verilog Assertions (SVA) offers constructs like "*eventuality*", "*until*" etc., collectively categorized as the "liveness properties". Roughly, a liveness property captures the essence of "something good will eventually happen" in contrast to a safety property which checks "Something bad will never happen in the system". For instance, when verifying a credit-based handshake between two agents, a property like "if all the credits are used, eventually some agent will return the credit back" can be expressed in SVA as:

!credit_available |-> s_eventually credit_available (P1)

Many commercially available FV tools can interpret and analyze this representation, but the associated FV complexity in verifying liveness properties presents significant challenges.

In literature, converting liveness to safety properties [1,2] has been proven to be quite useful, as multiple FV teams in the industry can attest. Some modern EDA tools also offer some impressive liveness bug-detection capabilities built on these translation tricks. However, due to the state space explosion inherent in model checking, these traditional methods have limitations in providing meaningful assistance beyond a certain point in the modern feature rich designs. This paper aims to bridge this gap.

The contributions of this work are two-fold. Firstly, we propose a method to decompose an end-to-end liveness property into smaller liveness properties in context of its Cone of Influence (COI) and a systematic way to stitch them together to ensure forward progress of the whole system. Second, we introduce a mathematically sound formulation for converting liveness properties into safety properties, with a focus on a particular "event" in the system, in contrast to the traditional "*k-liveness*" formulation [2], which doesn't scale well as the design complexity increases. Although several FV practitioners may have used this conversion in one form or another, we hope this work equips the readers with understanding to view the liveness properties in a more structured manner.

Our methodology has been applied to multiple multi-level multi-stage arbiters in the Last Level Cache (LLC) of Qualcomm Nuvia CPUs, each involving more than 40 requests sharing resources. Our method's decomposition strategy for smaller forward progress checks revealed multiple intriguing functional issues in these designs ranging from corner case performance issues to the system-level deadlocks.

II. MATHEMATICAL FORMULATION

In this section, we present some equations/properties, along with modeling code to build a foundation for the concepts discussed throughout the paper. Consider the following code snippet, which represents an event counter. The variable "*event_cnt*" is a flopped version of "*event_cntNxt*". *antc_seen* is designed to capture the phase when "*antc*" has been observed and "*consq*" has not been asserted yet. Upon *antc_seen* being asserted high, the counter counts the occurrences of *event*, resets to 0 when *consq* or *rst* is asserted high and hold it value otherwise.

```
assign event_cntNxt = (rst || consq)        ? {CNT_WID{1'b0}} :
                      (antc_seen && event) ? event_cnt + {{CNT_WID-1{1'b0}}, 1'b1} :
                                             event_cnt;
```

A. Traditional k-liveness

Consider the liveness property for an arbiter "every request to the arbiter must eventually be granted" which can be written as follows. Without loss of generality, we assume that "*req*" remains high till a "*gnt*" is observed throughout our discussions in this paper.

$$req \mid\!\!-\!\!> s\_eventually\ gnt \tag{P2}$$

P2 can also be represented as follows:

$$req \mid\!\!-\!\!> \quad \#\#[0{:}\$]\ gnt \tag{P3}$$

It should be noted that although P3 looks similar to pre-2009 liveness property syntax, in this paper, we are going to use P3 as a proxy for P2 for better readability in the context of mathematical formulation. In practice, use of P3 as SVA property should be avoided because according to the latest System Verilog standard [3], P3 is a weak liveness form and is always mapped to true without any regard for the antecedent and consequent expressions. Safety form of P3, traditionally referred as k-liveness in literature, can be written as follows when considering only a finite number of cycles instead of infinite horizon:

$$req \mid\!\!-\!\!> \#\#[0{:}k]\ gnt \tag{P4}$$

Interestingly, if we can prove that P4 holds for some finite k, it is not difficult to see that P3 will always be satisfied as well. Intuitively, if we can prove that "*gnt*" is observed within finite number of cycles, we can infer that "*gnt*" will eventually be asserted (given $k < \infty$). Unfortunately, calculating k for most designs is not trivial, making it challenging to apply this concept in its current form. Moreover, due to non-deterministic nature of formal analysis of inputs, finding a finite k is not always possible.

Let's re-write P4 by introducing a counter "*clk_cnt*" which starts from a reset value 0 when "*req*" is asserted and keeps incrementing for every posedge of clk until "*gnt*" is observed. It can be obtained by replacing (*antc = req, consq = gnt* and *event = 1'b1*) in the event counter. We can express P4 using "*clk_cnt*" as follows:

$$clk\_cnt \leq k \tag{P5}$$

B. "event" based k-liveness

Let's consider two more properties P6 and P7 below:

$$event\_cnt \leq k \tag{P6}$$

$$req \mid\!\!-\!\!> \#\#[0{:}\$]\ event \tag{P7}$$

"*event_cnt*" is implemented as a counter like "*clk_cnt*", but instead of counting posedges of "*clk*", it counts the number of times an "*event*" occurs in the design. The choice of "*event*" within the Cone of Influence (COI) of "*gnt*" should ensure some form of forward progress in the system. For instance, in a round robin arbiter which is enabled based on credit availability, "*event*" can be chosen to describe the condition when credits are available to the arbiter. In this case, P6 effectively checks that the round robin pick part of the arbitration scheme is functioning correctly, while P7 ensures that the resources eventually become available.

It can be demonstrated that if P6 and P7 hold for some {*event, k*} combination, then P3 will also always hold. Intuitively, if we can prove that "*gnt*" is seen in finite number of "*event*"s and further "*event*" occurs infinitely often, we can conclude that the "*gnt*" will occur infinitely often or eventually "*gnt*" will be asserted. This is a significant result and serves as a cornerstone of our proposed proof schema.

III. PROPOSED METHODOLOGY

We present an intuitive approach for verifying liveness properties in intricate designs. Our novel approach involves decomposing an end-to-end liveness property into a series of smaller, "*event*"-based safety properties, each establishing necessary conditions for forward progress. These intermediate safety properties serve as crucial building blocks in the form of helpers, ensuring eventual system progression.

A. Decomposition of liveness into {event, k} forward progress checker form

Figure 1. summarizes the steps for decomposing a liveness property into {*event*, *k*}-liveness and helper liveness properties. All formal tools return the result of a property as either full proof, failure (Counterexample - CEX) or inconclusive:

1. **Full Proof:** The property holds true for all possible legal stimulus.
2. **CEX:** Tool identifies a run of the system where property is falsified. The issue can be attributed to either a design bug (Real CEX) or an FV modeling issue (False CEX).
3. **Inconclusive:** In case of non-definitive (Inconclusive) results, the tool cannot find a bug nor able to prove its absence. Further analysis is required to extract useful information from the design in this case. It must be noted that bounded proofs in case of liveness properties are generally not as useful as bounded proofs of safety properties [4].

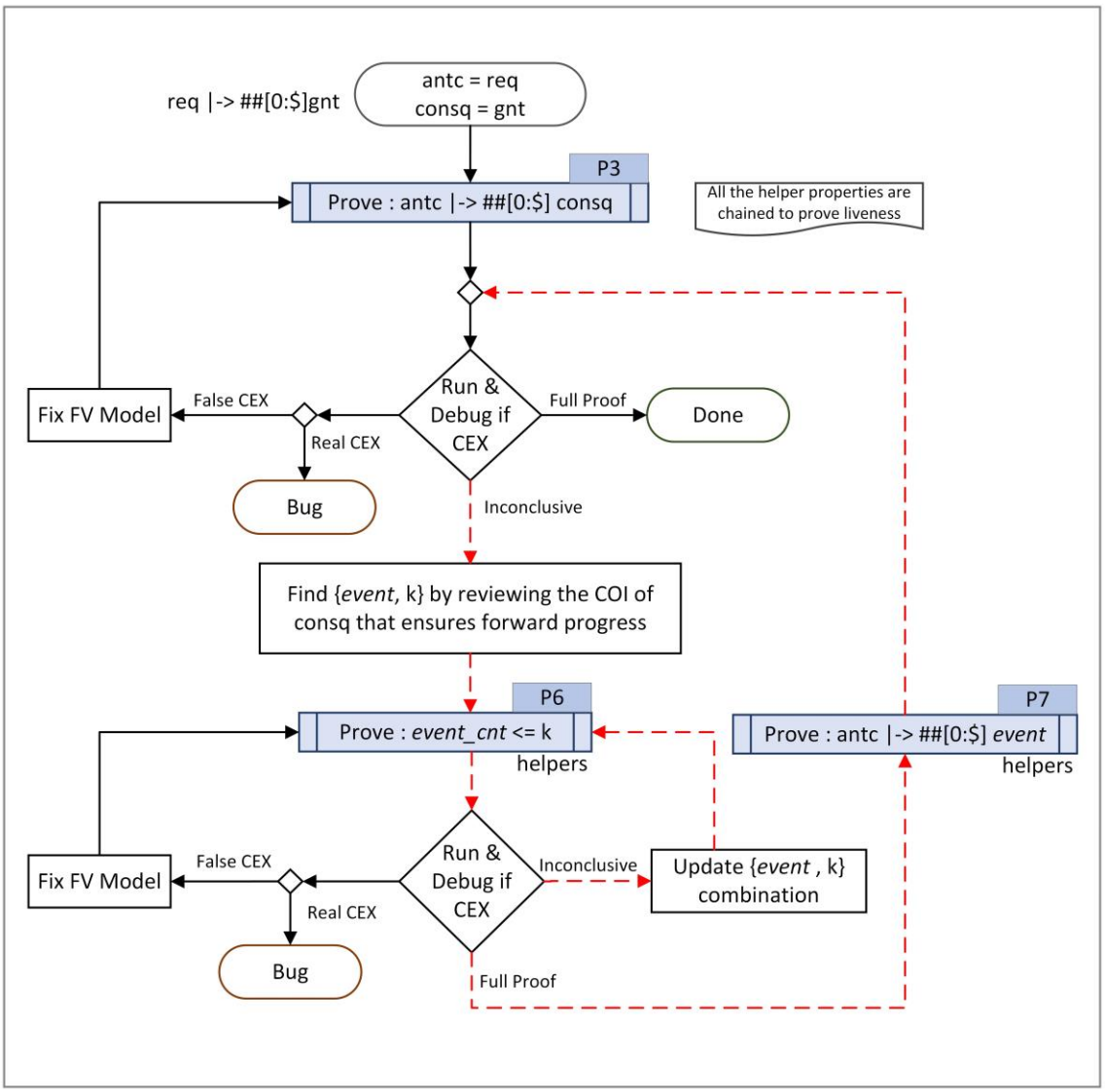

Figure 1. Decomposition of liveness property

The black flow lines in the Figure 1 highlight the concrete results of formal analysis and the dotted red flow lines represent the iterative steps to explore deeper into the design and if possible, prove the absence of liveness bugs. The first step is to define/choose an "*event*" in the COI of the P3, such that "consq" must be asserted within finite (≤ k) number of events. As P6 is a safety property, commercial EDA tools can reason more effectively than the P3. P6 can be further assisted by other "safety property" abstraction techniques. It's essential to note that defining a suitable "*event*" is not a trivial task; it requires fine-tuning and a basic understanding of the Design under Test (DUT). In our experience, selecting significant algorithmic decision points as "*event*"s have been quite fruitful as these signals (a) are less likely to change as the design evolves and (b) provides better communication with the design/DV teams. A case study in next section will illustrate this point further. Once a good {*event,k*} combination is defined, it is necessary to prove that the "event" is also making forward progress and a similar {*event,k*} analysis can be applied to P7. This approach can be iteratively applied until we reach leaf node(s) that can be proven in liveness form itself. All the properties developed in this approach can be used as helpers to prove end-to-end liveness property. As the COI of P7(s) is always expected to be a subset of P3, we have serendipitously observed that liveness bug-hunting tools provided by EDA vendors can identify issues much more efficiently on these intermediate properties.

B. Daisy chaining of helper properties

Consider an assertion module which can instantiate *a* liveness and *its* corresponding {*event,k*}-liveness property corresponding to P3 and P6 in Figure 2 respectively. If we map the steps described in previous section on to the FV_LIVENESS_PROP and instantiate the properties, it can be readily seen that a dependency chain will be formed between the properties similar to the Figure 3. Liveness property at each level uses the {*event,k*}-liveness at same level and the liveness property at a level below it as helper. For instance, *p_event1.liv* uses the *p_event2.liv* and *p_event1.kliv* as helpers. Another point worth mentioning is that Figure 1 can give an impression of linear dependency chains, but it can be non-linear in practice as single event may not be enough to capture the complete intent of the design, it has been illustrated in the Figure 3 for *event2* and *event3* nodes.

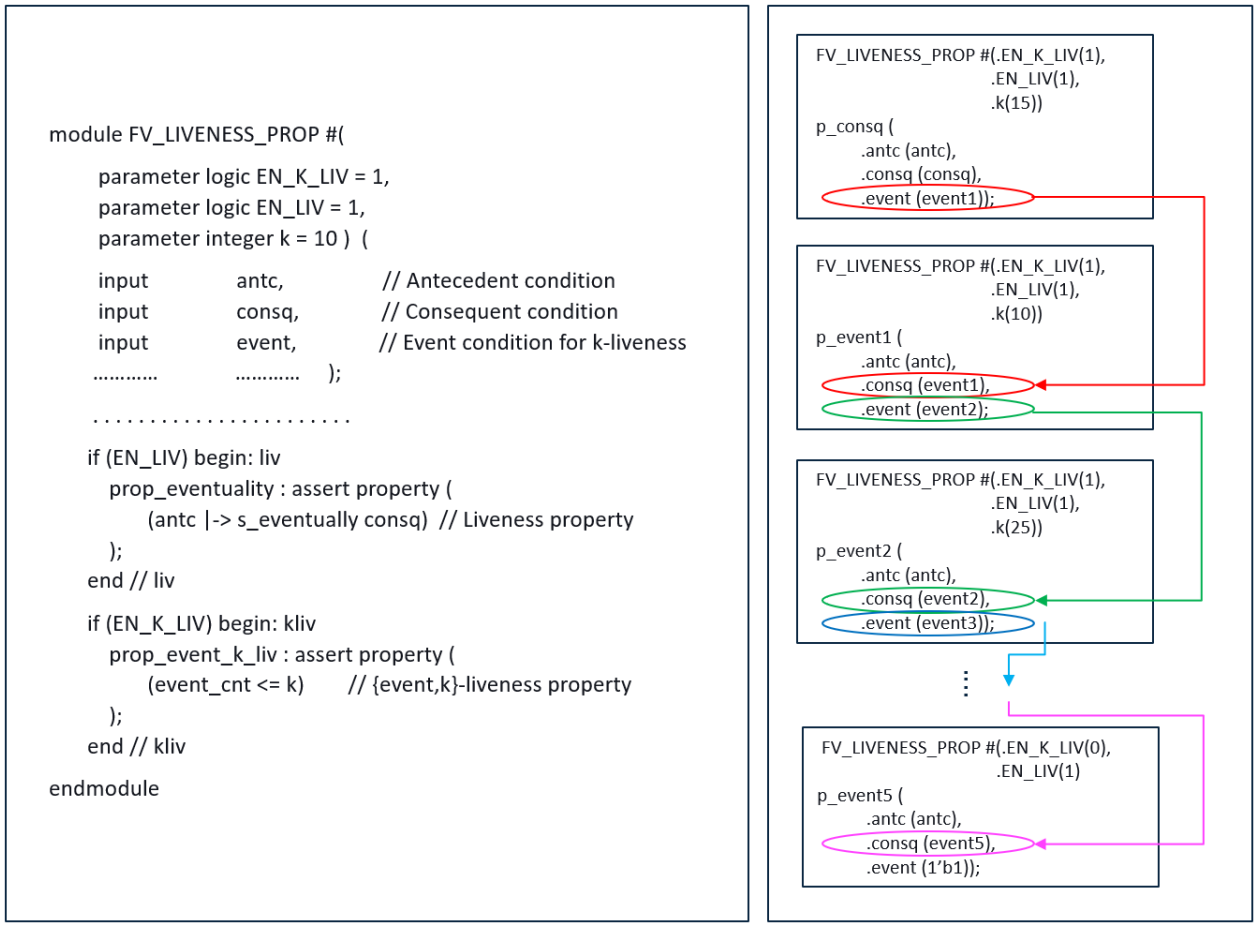

Figure 2. FV_LIVENESS_PROP

This approach seamlessly integrates Assume-Guarantee framework on intermediate safety properties which serve as the building blocks, ensuring forward progress and ultimately help in proving the original liveness property. It can leverage the advanced proof management tools provided by EDA vendors to easily define a proof decomposition tree and run all the leaves in parallel.

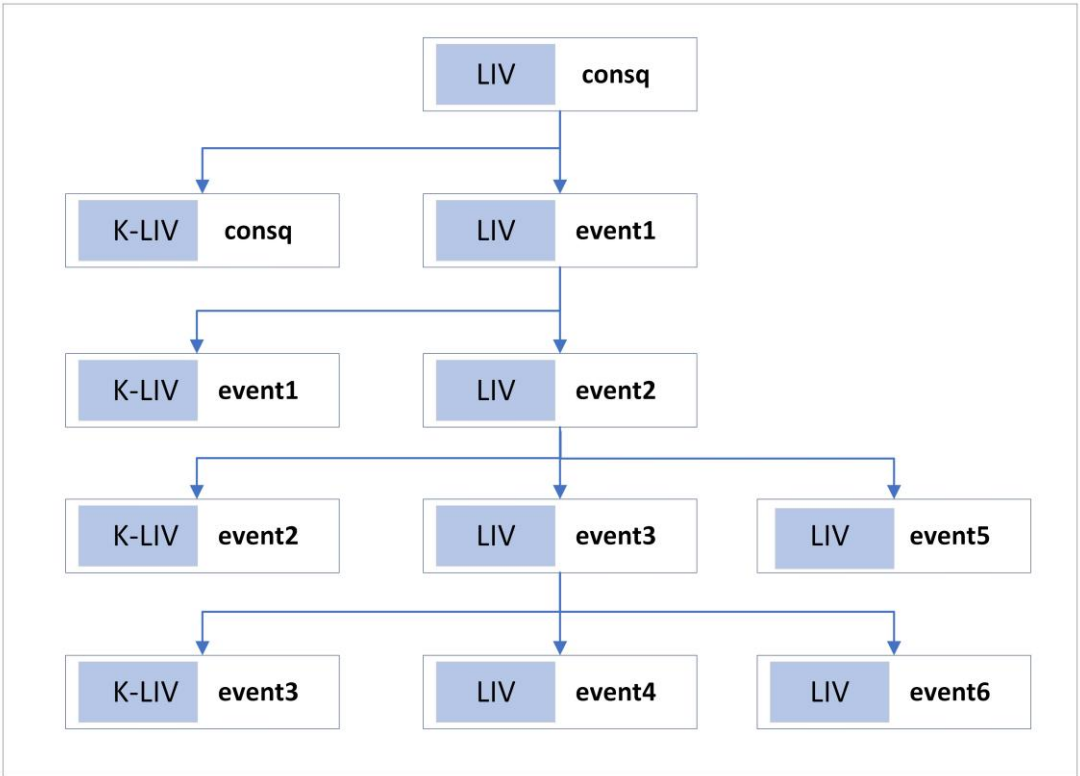

Figure 3. Daisy chaining of helper lemmas to build Assume-Guarantee Schema

IV. CASE STUDY

We implemented and tested our approach on multiple multi-level multi-stage arbiters in the Last Level Cache (LLC) of Qualcomm Nuvia CPUs, each involving more than 40 requests sharing resources. Our decomposition strategy for smaller forward progress checks revealed 6 intriguing functional issues in simulation signed off designs. We present simplified versions of two such bugs.

A. Multi-stage Arbiter

Figure 4 serves as an illustrative example of a multi-level arbiter. In this design, three distinct agents can send the requests to the arbiter. The requests occupy the dedicated out-of-order queues and wait for resources to be available to participate in the arbitration process (referred as “pickable” entry). The arbitration scheme checks for multiple picking rules like transaction priority rules, resource management etc. to decide which queue and entry within that queue can be serviced. It is essential to note that the local and global arbiters shown here are not simple fixed-priority or age-based fair arbiters but any entry within an agent’s queue becomes eligible for arbitration (pickable) if it satisfies the pre-defined picking rules for that agent. The complexity of this multi-stage arbiter grows exponentially with the number of agents involved and the intricacy of picking rule set. This intricate interplay of agents and pick rules adds to design complexity and introduces significant challenges in verification process.

The arbitration algorithm functions as follows. Firstly, the pickable entries of each agent participate in the local arbitration. The winners of local arbitration within each agent then advance to the global arbitration. An entry is granted if it wins at both local and global level. Interestingly, it is possible that an entry in a queue which was a local winner of round 1 and lost in the global arbitration may lose or not even participate in the local arbitration next time due to picking rules. There are dozens of picking rules mandated by the Spec that the arbitration scheme must respect. Here are few picking rules of this arbiter design:

- Exclusion rule: If any entry of a queue is granted, the queue can't participate in the global arbitration for the next 3 cycles. This exclusion period is essential to facilitate the return of credits.
- Bank matching rule: Two requests targeting same bank cannot be granted in two adjacent cycles to avoid data array bank conflict.

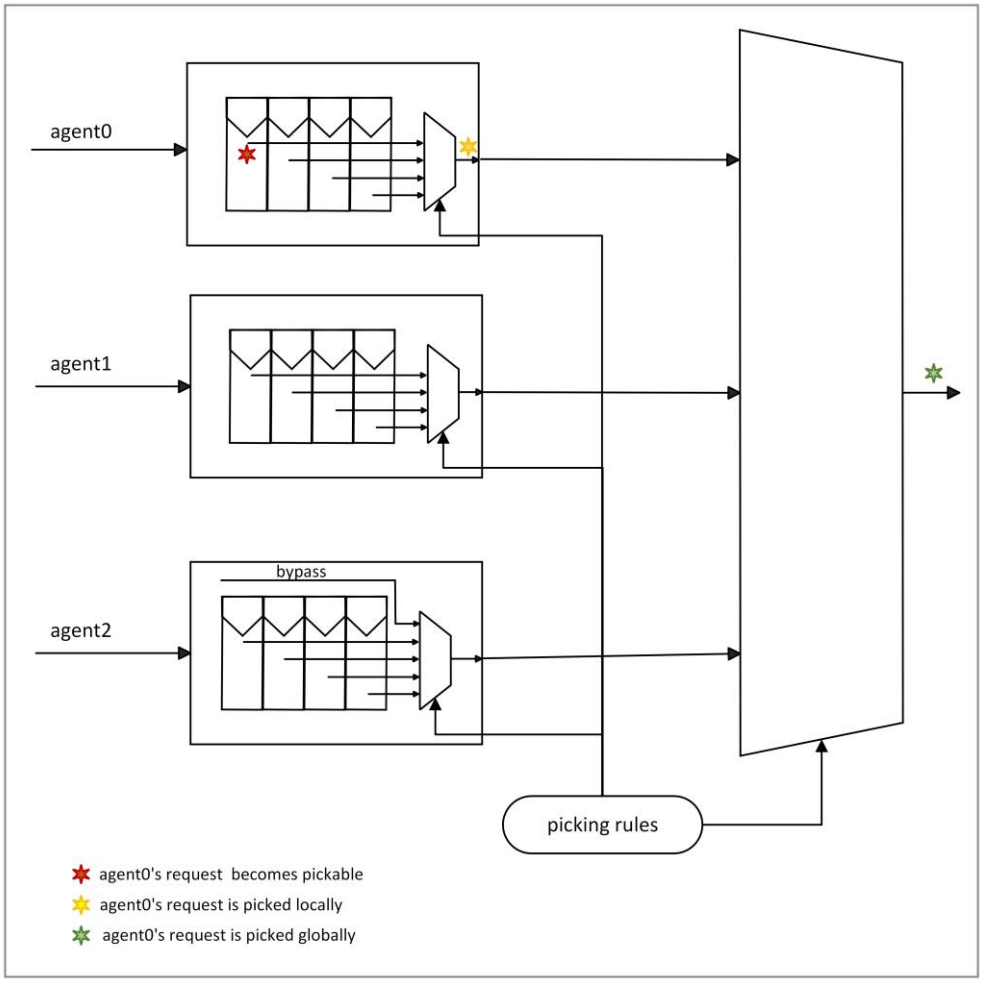

Figure 4. Multi-stage arbiter

B. Implementation

To ensure that the design always makes forward progress, we need to prove that any request made by an agent should eventually be granted. Let's examine the case of agent0's entry0. The major decision points for agent0's entry0 are highlighted in Figure 4 which are the global pick, the local pick, and the resource availability.

Table 1 serves as a reference to understand the sequence of events in this arbiter design which ensures forward progress in proving that *agent0_entry0_req* is eventually getting granted. While *event1* and *event2* in the table correspond to global pick and local pick respectively, resource availability is captured by event3 and event 4.

antc	agent0_entry0_req
consq	agent0_entry0_gnt
event1	any_agent0_req_got_picked_globally
event2	any_agent0_req_got_picked_locally
event3	agent0_entry0_is_pickable
event4	interface_credits_avilable

Table 1. Assume-Guarantee events

Let's expand some of the properties of this table. *consq.kliv* asserts that given adequate number of agent0 picks globally, entry0 of agent0 must be granted. event4.liv checks resources/credits are eventually available from the interface. As *event4* is close to the DUT's primary inputs and has small COI, it is highly likely that FV tools would be able to prove the liveness property directly and we won't need to write the corresponding k-liveness. These properties are stitched together in the Assume-Guarantee Schema like Figure 3. In addition to initial liveness property, this example generates 4 liveness and 4 {*event,k*}-liveness properties which act as helpers to prove the end-to-end liveness property. It is worth mentioning that in this implementation, we can directly work with the interface fairness assumptions in contrast to the traditional over-constraint safety assumptions to work with complex liveness environments.

C. Results

1. Starvation issue

We found multiple scenarios in the arbiter where the harmonic interplay of various picking rules leads to starvation of a particular entry in the queue. These are classic live locks where the system gives the illusion of making progress, but parts of the system are denied service indefinitely. Figure 5 illustrates one such scenario where *agent0_entry0_req* with TID 'B' and targeting bank B2 is getting indefinitely starved due to alternative restrictions imposed by "Exclusion rule" and "Bank matching rule".

$$agent0\_entry0\_not\_pickable = agent0\_entry0\_exclusion \mid agent0\_entry0\_bnk2\_mtch$$

The liveness CEX trace reveals a 4-cycle loop that can repeat indefinitely. *agent0_entry0_exclusion* is asserted for 3-cycles (Exclusion rule) as agent0 request with TID 'A' is granted globally and *agent0_entry0_bnk2_mtch* is asserted for 1-cycle (Bank matching rule) as agent_1 targeting bank B2 is granted globally. This repetitive pattern can cause the *agent0_entry0_not_pickable* to stay asserted forever making the agent0 request with TID 'B' targeting bank 'B2' starve indefinitely.

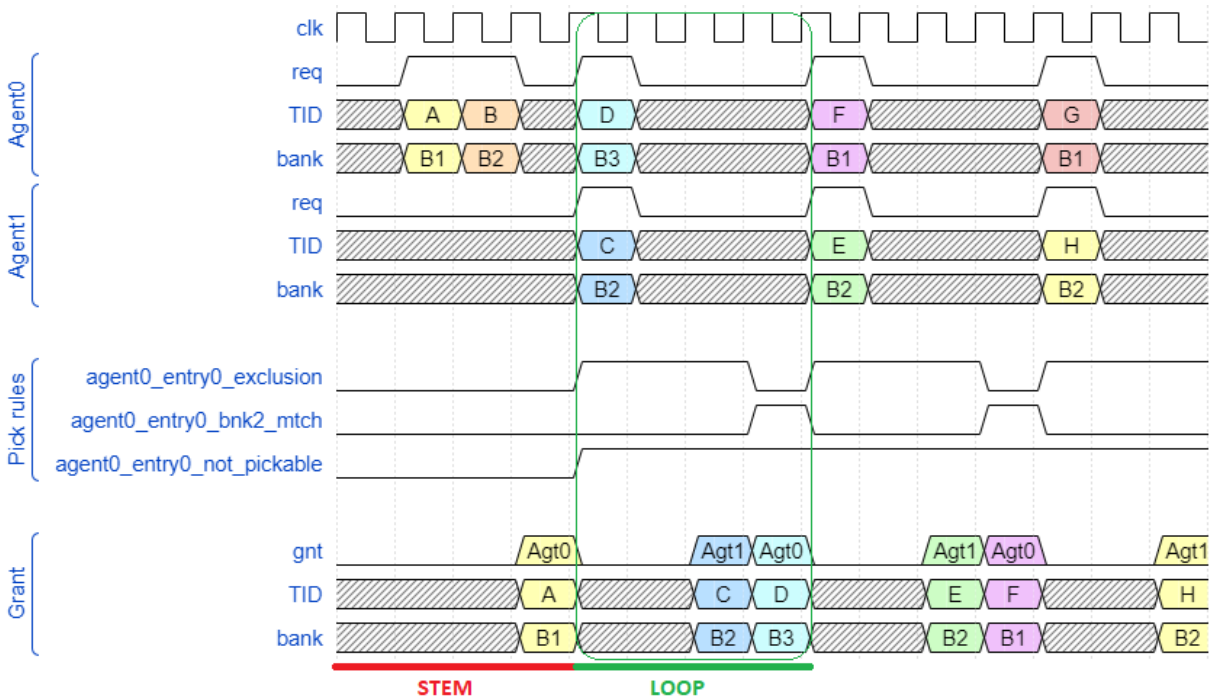

Figure 5. Waveform depicting a starvation case

The "Exclusion rule" is effectively denying agent0 to participate in the global arbitration process allowing agent1 to use bank2 which will trigger "Bank matching rule" for next 1 cycle. The "Bank matching rule" on the other hand can be exploited by some different entry of agent0 (by targeting non-B2 bank) blocking any pick from agent0 for next 3 cycles.

2. *Deadlock issue*

In the upcoming generation of our design a new picking rule has been introduced, focusing on priority of the requests. This rule stipulates that:

- Priority rule: Any request marked as low priority can only use resources from the common pool while a request marked as high priority can use common as well as the resources reserved for requesting agent.

This setup has unveiled a system-level deadlock scenario. It arises when a high-priority request is pending behind a low-priority request. To account for starvation class of bugs from the design, an anti-starvation widget was introduced in the arbiter control which restricts the local picks of all the entries from an agent except the one which has marked itself as the "oldest starved". This anti-starvation widget along with new priority rule led to a critical corner case deadlock. If a low-priority request marked itself as "oldest starved", it can prevent the grant of all the younger entries from the same agent including the high priority requests. Simultaneously, it is possible that the issuing agent is waiting for the high priority to be granted to process the low-priority request. This creates a circular dependency between the agents and arbiter leading to a system level deadlock.

V. CONCLUSION AND FUTURE ASPECTS

Ensuring forward progress of a system is critical for modern hardware designs. In this work, we proposed a novel approach to decompose an end-to-end liveness property into smaller liveness properties in context of its Cone of Influence (COI) and a systematic way to stitch them together to ensure forward progress of the whole system. In the process, we found multiple intriguing issues which were missed by traditional dynamic simulation and even end-to-end formal verification. Consequently, design team addressed these issues and was able to close the verification efforts with high quality sign-off. Additionally, the bugs found, and the safety properties developed during this analysis can be leveraged by simulation teams for improving their stimulus.

This method has the potential to be used on a wide variety of designs like the FSMs (desirable state is eventually reached/transitioned from), LRU (every entry has a path to LRU), Network-on-Chip (packets reach destination without being dropped or delayed indefinitely), etc. Additionally, we envision discovering more efficient methods around {event, k} selection in near future.

REFERENCES

[1] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science, 66(2):160–177, 2002.

[2] Koen Claessen and Niklas S¨orensson. A liveness checking algorithm that counts. In 2012 Formal Methods in Computer-Aided Design (FMCAD), pages 52–59. IEEE, 2012.

[3] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800- 2017 (Revision of IEEE Std 1800-2012), vol., no., pp.1-1315, 22 Feb. 2018.

[4] Kim, NamDo, Junhyuk Park, HarGovind Singh, and Vigyan Singhal. "Sign-off with Bounded Formal Verification Proofs." In Design and Verification Conference. 2014.