

Achieving Full Liveness Proofs via a Systematic Assume-Guarantee Approach and Iterative Helper Generation

Stella Simic, ssimic@qti.qualcomm.com, Qualcomm Inc.
Karthik Baddam, kbaddam@qti.qualcomm.com, Qualcomm Inc.

Abstract — Proving the liveness of a design is notoriously difficult, as this requires an analysis of infinite behaviors. We outline a strategy for achieving full liveness proofs through formal verification by translating a liveness property into a series of simpler properties. By considering a chain of safety properties placed in an assume-guarantee structure, and by iteratively adding helpers to reduce the state space for non-converging assertions, we achieve convergence on all newly created properties. The proof of the original liveness property then follows as a trivial consequence. We applied our strategy on a formal test bench for an FSM within the MMU of the next generation Qualcomm Snapdragon compute CPU in which several liveness checks were not converging with a time-out of 7 days. Using our approach, we were able to achieve a full liveness proof in a matter of hours.

I. INTRODUCTION

Proving liveness-related properties such as the absence of deadlock or livelock [4] and forward progress is known to be a difficult task, as this entails the analysis of infinite traces. While formal verification can, in theory, give us a definitive answer, in practice it is often hard to reach convergence on a liveness proof. In real-world hardware designs, proving liveness of a finite state machine (FSM) requires extensive effort, as transitions between states are often governed by highly complex logic. Our work is centered on a block within a high performance, superscalar flagship Qualcomm CPU core, where we approach the following problem: proving forward progress on an FSM related to translation invalidation, starting from a non-converging liveness property. We use Cadence’s Jasper tool for formal verification of the case study presented in this paper, but we believe that the ideas presented here can be applied using other commercially available formal verification tools such as Synopsys’s VC Formal and Siemens EDA’s Questa Formal.

We address the liveness problem by leveraging several recent approaches for proof simplification. First, we translate liveness checking into safety checking by considering k-liveness formulations [1,2,7]: we use an event-based k-liveness approach and create a chain of safety properties, and an assume-guarantee based reasoning. We then perform a form of reset abstraction on any undetermined properties to find and add helper assertions to reduce the state space. This form of reset abstraction is available in most commercial FV tools; Jasper calls this feature state-space tunnelling (SST) [3]. Finally, when SST does not provide any more clues, we use a simple automatic helper generation approach. Our overall approach consists in systematically employing the power of an assume-guarantee decomposition and using state-space reduction techniques until we reach convergence on all properties.

This paper is organized as follows. In Section II. we introduce all the technical concepts needed to understand our approach, namely, k-liveness, assume-guarantee and SST. In section III. we outline the application of these techniques to our case study. Section IV. summarizes the results and findings.

II. METHODOLOGY

In the rest of this paper, we assume familiarity with SystemVerilog Assertions (SVA) [5,6]. We will express properties in the following form: “[assertion/assumption name] : [expression]”, using intuitive naming for future referencing. When using the construct “##[0:\$]” we mean s_ eventually (strong eventuality).

Consider the following property stating that, starting from a non-reset state of an FSM, we will eventually reach the FSM’s reset state:

$$\text{asrt_liv_fsm: (state != reset_state) |-> ##[0:$] (state == reset_state).} \quad (1)$$

A direct check of such a liveness assertion will rarely converge and we need to simplify this problem. In the following we describe how to break a liveness check into simpler checks and how to reduce the state space that needs to be considered.

A. Liveness as safety

Let us focus for now on a single non-reset state S of an FSM and on proving forward progress for this state. We consider the following assertion, stating that once the FSM is in state S , it will eventually leave state S :

$$\text{asrt_liv_S: } (\text{state} == S) \mid\text{-->} \#[0:\$] (\text{state} != S) . \quad (2)$$

A possible approach to proving liveness as in (2) is to use an event-based k -liveness formulation of this check. This consists in finding a relevant event in the cone of interest (COI) of (2) and a natural number k and proving that: a) given the antecedent, the consequent will occur in at most k occurrences of the event, and b) the chosen event does eventually occur. This breakdown of the original liveness check consists of proving two new properties: a safety property and a liveness property (now with a smaller COI).

The translation of (2) into a k -liveness property is therefore performed as follows. We need to look for a relevant event “evnt” and a number $k1$ and try to prove that the number of occurrences of evnt, while the FSM is in state S , is less than $k1$. This is equivalent to proving that the number of cycles spent in S while evnt also holds true is bounded by $k1$. If we are also able to prove that evnt eventually occurs, then the proof of (2) follows as a consequence. Indeed, the following two properties:

$$\text{asrt_b_liv_S_w_evnt: } \text{count}(\text{evnt}, S) \leq k1 \quad (3)$$

$$\text{asrt_liv_evnt: } \sim\text{evnt} \mid\text{-->} \#[0:\$] \text{evnt} . \quad (4)$$

imply that (2) also holds. We may even be able to prove a bounded (stronger) formulation of (4), i.e.,

$$\text{asrt_b_liv_evnt: } \sim\text{evnt} \mid\text{-->} \#[0:k2] \text{evnt} , \quad (5)$$

for an adequate bound $k2$. In that case, Eq. (3) and (5) are both safety properties.

B. Assume-guarantee

Self-loops are often the bottleneck of FSM liveness checks. Once state S is entered, a certain blocking event A may need to occur N times before S is left. An example of event A is an index that needs to go through a set of N possible values. We can then consider the following k -liveness property stating that, once the FSM is in state S , then event A happens in at most $k3$ occurrences of evnt:

$$\text{asrt_b_liv_A_w_evnt: } \text{count}(\text{evnt}, (S \ \& \ \sim A)) \leq k3. \quad (6)$$

If we are able to prove (6), we can then assume this to be true and use it as an assumption and try to prove that the maximum number of occurrences of A will happen in at most $N*k3$ cycles:

$$\text{asrt_b_liv_maxA_w_evnt: } \text{count}(\text{evnt}, (S \ \& \ \sim \text{maxA})) \leq N * k3 . \quad (7)$$

Once (7) is proven, we can use it as an assumption for the proof of (3). In general, we may add as many properties as necessary to reach a proof of (3).

Using an EDA tool, we can organize our properties into nodes in the order they need to be proved (guaranteed) and later assumed. The first node will contain properties that do not require any assumptions, the second node will contain properties that rely on assumptions in the first node, and the last node will contain the most complex properties which require all other properties to be assumed. FV tools often provide features that help organize assume-guarantee structures; in Jasper this feature is called Proof Structure.

C. Helper generation

A helper or invariant is a property that always holds true for a design. Ideally, a helper is a simple statement which describes allowed behaviors, thus excluding infeasible behaviors from being considered. When such a property is proven, it can be used as an assumption to help prove more complex properties. Helpers can often be identified through human analysis of the DUT. For example, simple embedded RTL assertions already provided by the designers may be considered helpers. Additionally, one may come up with properties simply from their knowledge of the design. Examples are: a given signal is onehot or a given counter can only increase its value by one.

When we are faced with complex non-converging properties, we may try to generate helpers to reduce the state space that the verification tool needs to explore to reach a proof. If we are not able to manually create helpers, we can turn to SST to help in finding helpers. For a given non-converging property and a selected bound M , an SST trace will show a failure starting from an abstracted initial state, which might not be reachable, and ending in a failure in M cycles. Through this analysis, we hope to find insights into impossible combinations of variable values and add helper properties stating that such scenarios may never happen.

For example, consider two signals, `signal1` and `signal2`. We suspect that only one signal can be asserted at a time. However, the SST trace shows a cycle in which `signal1 = 1'b1` and `signal2 = 1'b1`. We think this is not possible, so we check the following property:

$$\text{asrt_helper1: } \sim(\text{signal1} \ \& \ \text{signal2}). \quad (8)$$

If this property is proven, then we can assume it to reduce the state space and simplify state exploration during the proof of our non-converging property.

If SST analysis does not provide any more clues, we may resort to automatic helper generation. Given an SST trace and a list of signals that we suspect may have interesting relations (but are unable to formulate those relations into a property), we use a script to automatically generate a helper listing all valid value combinations for these signals. The script is an implementation of the technique described in [3].

For example, consider two 3-bit signals, `process_done` and `process_in_progress`. Each bit in these signals is asserted when the corresponding process is done or is in progress, respectively. If we know that these signals are involved in a non-converging property but relating them is not obvious, we use the automatic helper generation script to generate all possible value combinations for this list of two signals. Consider a result in the following form:

$$\begin{aligned} & ((\text{process_done} == 3'b000) \wedge (\text{process_in_progress} == 3'b000) \vee \\ & (\text{process_done} == 3'b000) \wedge (\text{process_in_progress} == 3'b001) \vee \\ & (\text{process_done} == 3'b000) \wedge (\text{process_in_progress} == 3'b010) \vee \\ & (\text{process_done} == 3'b000) \wedge (\text{process_in_progress} == 3'b100) \vee \\ & \dots \\ & (\text{process_done} == 3'b001) \wedge (\text{process_in_progress} == 3'b000) \vee \\ & (\text{process_done} == 3'b001) \wedge (\text{process_in_progress} == 3'b010) \vee \\ & (\text{process_done} == 3'b001) \wedge (\text{process_in_progress} == 3'b100) \vee \\ & (\text{process_done} == 3'b011) \wedge (\text{process_in_progress} == 3'b000) \vee \\ & (\text{process_done} == 3'b011) \wedge (\text{process_in_progress} == 3'b100) \vee \\ & (\text{process_done} == 3'b111) \wedge (\text{process_in_progress} == 3'b000)). \end{aligned}$$

After some inspection, we notice that a process can only be in progress if it is not already done and that only one process can be in progress at a time. When used as a helper, `asrt_helper2`, the expression above reduces the state space significantly, as it only allows 20 feasible combinations instead of the possible $2^6 = 64$.

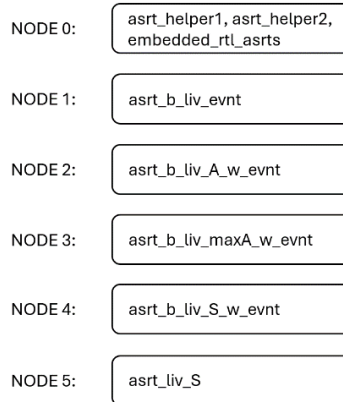


Figure 1: Assume-guarantee nodes. The simplest assertions (helpers and embedded assertions) are placed in node 0 and the most complex assertion (forward progress in state S) is in node 5. In particular, the proof of node 5 follows from the assumption of nodes 4 and 1.

D. Overall approach

Recall the liveness assertion we considered at the beginning of this section, namely, asrt_liv_S in Eq. (2). To reach convergence on the proof of this property, we broke it down into a chain of simpler properties. Putting all of the above together, we show the assume-guarantee nodes in Figure. 1.

III. CASE STUDY

Our DUT is an invalidation FSM within the MMU of a Qualcomm CPU core, as shown in Figure. 2. It receives an invalidation request from an invalidation queue and accesses every entry in a RAM, ensuring its contents are invalidated if required. Once done, it sends out a done signal. The FSM has 6 states, two of which are self-looping. The reset state of the FSM is IDLE. The goal is to prove that state DONE is always eventually reached, i.e.,

$$\text{asrt_liv_InvalFsm: InvalReq} \mid\!\!\rightarrow \text{##}[0:\$] \text{InvalDone}. \quad (9)$$

A direct proof attempt on this assertion did not converge with a timeout of one week. Hence, we applied the strategy described in Section II. In the rest of this section we illustrate how we achieved this.

From our knowledge of the DUT, we knew there is an event in the COI of (9) that is needed for forward progress, as it was involved across the FSM logic. The event was $\text{evnt} = (\text{InvReqVld} \ \& \ \sim\text{InvLkupGnt})$, or, in natural language: “the invalidation request is valid but there is no invalidation lookup grant”. In the case of this DUT, the logic behind these signals was entirely internal to the DUT, and we were able to figure out a bound $k_{\text{evnt}} = 4$ on the number of cycles it takes for evnt to occur. In other words, we were able to prove:

$$\text{asrt_b_liv_evnt:} \sim\text{evnt} \mid\!\!\rightarrow \text{##}[0:4] \text{evnt}. \quad (10)$$

We then attempted to prove the following event-based k -liveness property, with a reasonable (conservative) bound $k_{\text{fsm_liv}} = 650$ which we guessed according to our knowledge of the DUT:

$$\text{asrt_b_liv_InvalFsm_w_evnt: count}(\text{evnt}, \sim\text{InvalDone}) \leq 650, \quad (11)$$

which, in natural language, means “InvalDone will be asserted in at most 650 occurrences of evnt , after the InvalReq signal was seen”. This proof did not converge, so we decided to focus on proving forward progress on the single states of the FSM. State S3 proved to be the most complex to reason about, hence we will outline how we approached it.

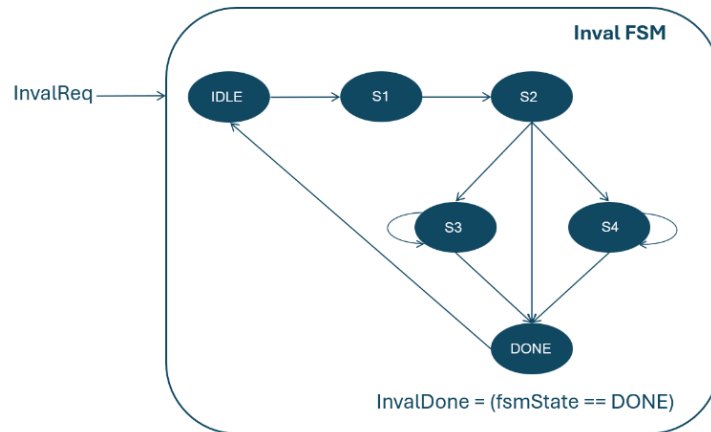


Figure 2: Invalidation FSM. Once an invalidation request InvalReq is received, the FSM processes it and should eventually issue a done signal, InvalDone.

From our knowledge of the design, we knew that,

1. once the FSM has entered state S3, it will stay in S3 while a signal called S3Done == 0;
2. S3Done == 0 while an index called idx hasn't reached its maximum value, i.e., $idx < idx_max$;
3. the initial value of idx is 0 and $idx_max == 127$;
4. idx increases by one when the grant signal is asserted: $InvLkupGnt == 1'b1$.

Notice that point 3. above implies that idx can increase at most 128 times. Our goal was now to prove:

$$asrt_b_liv_S3_w_evnt: \text{count}(\text{evnt}, S3) < 261, \quad (12)$$

meaning “once the FSM enters state S3, it will leave S3 in at most 261 occurrences of evnt”. Our estimate of 261 occurrences of evnt was given as double the amount of index increases that need to happen, plus 5 extra occurrences of evnt, as an overestimate. This property was not converging either, so we decided to look more into detail at the events that need to occur for the FSM to leave S3.

We decided that it would be useful to prove that the index idx does eventually increase when the FSM is in state S3, in at most $k_idx_liv = 2$ occurrences of evnt:

$$asrt_b_liv_idx_w_evnt: \text{count}(\text{evnt}, idx) \leq 2, \quad (13)$$

meaning “once the FSM is in state S3, the index increases in at most 2 occurrences of evnt”. Now this property converged and we managed to prove it. We then tried to prove $asrt_b_liv_S3_w_evnt$ in Eq. (12) again, using $asrt_b_liv_idx_w_evnt$ as an assumption. Notice that the estimate 261 that we chose was now confirmed to be sensible, as $261 = 2 * 128 + 5$, in other words, two evnt occurrences per idx increase plus an extra 5. This assertion still did not converge, so we decided to perform SST on it.

The 15-cycle trace our FV tool provided is depicted in Figure. 3., in a simplified version. Here, we have plotted the interesting signals that we suspected would be the culprits of the false failure. In this waveform, the signal count is the counter in Eq. (12). We can see that the failure in cycle 15, consisting in the count exceeding the bound 261 in Eq. (12), is due to the initial value of count in cycle 1 not being possible in combination with the initial value of idx. Indeed, the index increases in the worst case with every two occurrences of evnt, but here we have $idx = 124$ and $\text{count} = 256 > (124 + 1) * 2$. We add 1 in the right-hand side of this equation as the minimum value of idx is zero.

We therefore wrote a property to exclude such behaviours, namely:

$$asrt_helper_count_S3_bounded: \text{count}(\text{evnt}, S3) \leq (idx + 1) * 2. \quad (14)$$

We first proved this property and then used it as a helper to prove the k-liveness properties considered earlier. We also added a number of other helpers, such as:

$$asrt_helper_count_S3_reset: (state \neq S3) \rightarrow (\text{count}(\text{evnt}, S3) == '0), \quad (15)$$

meaning “the counter for occurrences of evnt in state S3 is zero when the FSM is not in S3”. We kept adding helpers until all of our properties converged. The final proof structure was organized as depicted in Figure. 4.

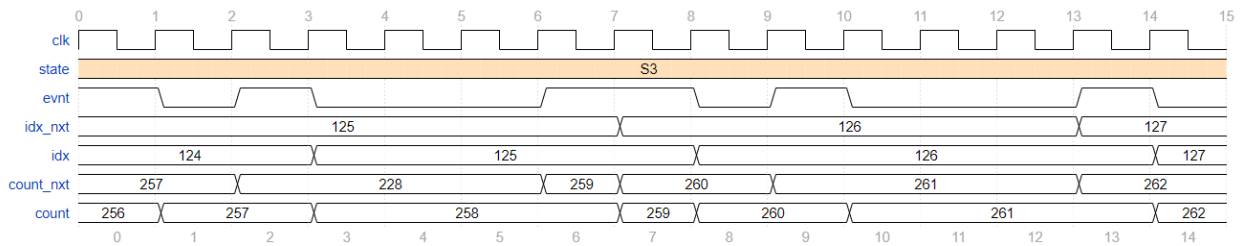


Figure 3: Waveform illustrating SST trace for $asrt_b_liv_S3_w_evnt$. Infeasible combination of values for count and idx in cycle 1, resulting in false failure in cycle 15.

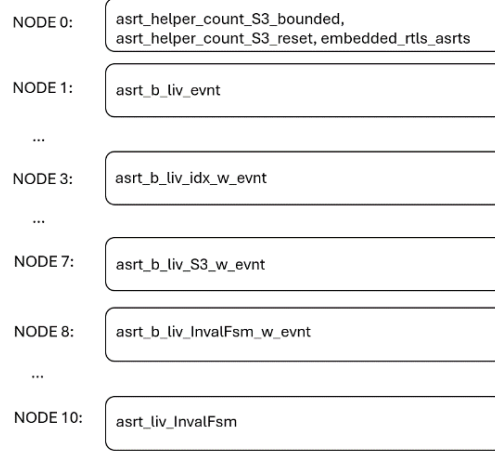


Figure 4: Final assume-guarantee nodes. Only showing nodes with properties regarding S3 forward progress and end-to-end liveness properties. All other properties regarding other states are not shown here, but follow similar reasoning.

In node 0 of the assume-guarantee nodes, we placed all the embedded RTL assertions provided by the designers, and all of the helpers, such as `asrt_helper_count_S3_bounded` and `asrt_helper_count_S3_reset` (Eq. (14) and (15), respectively). In node 1 we placed the bounded liveness (safety) property `asrt_b_liv_evt` (Eq. (10)). In node 3 we placed `asrt_b_liv_idx_w_evt` from Eq. (13). Node 7 contains `asrt_b_liv_S3_w_evt`, from Eq. (12). Node 8 contains `asrt_b_liv_InvalFsm_w_evt`, from Eq. (11). This is the event-based k -liveness property proving end-to-end liveness for the entire FSM. Finally, we placed the full liveness assertion, namely `asrt_liv_InvalFsm` from Eq. (9) in node 10.

The results of this strategy were very encouraging. We managed to prove all properties in our assume-guarantee nodes in a matter of hours. Most notably, node 7 took ~20s, node 8 took ~7h and node 10 took ~7min. Recall that, without all of the previous nodes (0-9), the property in node 10 was timing out after 1 week of runtime. Figure. 5. shows a screenshot of our assume-guarantee nodes captured in Jasper’s proof structure window.

IV. RESULTS AND FUTURE WORK

In this paper, we have outlined a strategy to tackle non-converging liveness checks on FSMs with self-looping states. The strategy includes two main techniques of proof simplification. The first is a break-down of liveness properties into a chain of safety properties focusing on smaller portions of the FSM. The second is a reduction of the state space by way of helper assertions.

We validated our approach on a complex invalidation FSM. Our approach proved to be successful, as we were able to reach a full liveness proof in a matter of hours, while earlier our checks were inconclusive even after a week of run time. We believe that our approach carries over well to other designs, as many of our strategies are universal across FSMs. In particular, the critical event for k -liveness and the repetitive events for self-looping states are often of the same form. Moreover, our intuition is that focusing on counters and indices and their mutual relations is a good rule of thumb. We are testing this conjecture on a more complex case study and have gotten promising results.

While good helpers can dramatically reduce the state space and help reach proofs on complex liveness properties, we also found that finding a helper, be it from an SST trace or otherwise, for a k -liveness property often requires extensive human effort. A good intuition, stemming from deep knowledge of the DUT is often needed. Even then, SST trace analysis can hit a dead end in helper generation. Moreover, if the expression used in a helper assertion is a complex Boolean formula, this may even be detrimental to the proofs of properties in subsequent nodes of the assume-guarantee chain.

In conclusion, generating strong helpers is crucial for reaching convergence and our current work on other case studies shows that this is a hard task and that not all helpers are helpful. We believe that the field of machine learning and AI has a role to play in this task, and we are exploring potential ways forward in this regard. Our experience with the DUT presented here has given us some insight into interesting classes of signals to focus on and relations to look for when considering two counter type signals. This may be used as a template for learning new helpers.



























Proof Structure					
Name		Type		PS Result	Propagation
 LIVENESS_AG		Assume Guarantee		38:1503:0	All
 EMBEDDED_AG.G0		imp(guarantee)		22:0:0	All
 LIVENESS_AG.G1		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G2		imp(cumulative-AG)		1:0:0	All
 LIVENESS_AG.G3		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G4		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G5		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G6		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G7		imp(cumulative-AG)		2:0:0	All
 LIVENESS_AG.G8		imp(cumulative-AG)		1:0:0	All
 LIVENESS_AG.G9		imp(cumulative-AG)		1:0:0	All
 LIVENESS_AG.G10		imp(cumulative-AG)		1:0:0	All
 NONLIVENESS		imp(assume)		0:1503:0	All

Figure 5: Screenshot of our assume-guarantee nodes captured in Jasper’s Proof Structure window.

REFERENCES

- [1] N. Sharma, V.N. Narisetty, “Liveness Assume-Guarantee Proof Schema: A Step Towards Liveness Full Proofs”, DVCON-US, March ‘24.
- [2] K. Claessen, N. Sörensson, “A Liveness Checking Algorithm that Counts”, FMCAD, ‘12.
- [3] D. Gilday, “Human-Guided Proof Closure”, Jasper User Group, November ‘21.
- [4] A. Garg, “Forward Progress Checks in Formal Verification: Liveness vs Safety”, DVCON-US, March ‘24.
- [5] E. Seligman, T. Schubert, M.V.A.K. Kumar, “Formal Verification – An Essential Toolkit for Modern VLSI Design”, 2nd ed., Elsevier, ‘23.
- [6] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language,” IEEE STD 1800-2009, Dec ‘09.
- [7] A. Biere, C. Artho, V. Schuppan, “Liveness checking as safety checking”. ENTCS, 66(2):160–177, ‘02.