Forward Progress Checks in Formal Verification: Liveness vs Safety

Ankit Garg
NVIDIA, Santa Clara, CA
ankigarg@nvidia.com

***Abstract*- This paper focuses on the different techniques and methodologies employed to analyze and prove forward progress and its significance in the context of formal verification.**

We will discuss various approaches for ensuring forward progress in the design, using both Liveness and Safety assertions and later examine how and where each assertion type should be used. Some case studies will be used to demonstrate the application of forward progress checks and the types of critical issues it has found in the designs.

I. Introduction

Formal verification plays a crucial role in today's design verification flows. There's no doubt that formal verification does tasks which are extremely difficult in simulation-based verification. One such task is checking forward progress in a System, ensuring that a system eventually reaches a desired state or completes its intended behavior. Forward progress verification guarantees that a system continuously makes progress and avoids getting stuck in an undesirable or deadlock state.

This problem becomes particularly significant when dealing with critical design components in high performance computing systems. These systems interact with multiple internal/external controllers (Fig. 1), where it's not always guaranteed that the drivers will strictly align to specification. The error reporting mechanism in these systems is designed to detect and report instances of incorrect input. However, if the system experiences a freeze due to erroneous input, a comprehensive system reset is usually required to restore its functionality. System resets are undesirable because it can result into data loss, service disruptions and impacts the users working on it.

Figure 1. Controller (left) in complete hang with the system (right)

Forward progress testing is extremely critical here to make sure the above situation never happens in a live system.

II. Understanding Forward Progress

Forward progress ensures that a system or design consistently advances toward its intended goal, avoiding stagnation or undesirable states. The source of unexpected stagnation could be either in software or hardware.

It's extremely critical to ensure hardware design isn't causing a system hang. For that we need to identify sources of possible hangs in the RTL design. We can roughly classify most of these into three categories.

A. Deadlock

Deadlock happens when a cyclic sequential logic (eg: FSMs, Counters) get stuck in a specific state indefinitely. The diagram below (Fig 2) shows an FSM (Finite State Machine) which is stuck in a non-reset state (State=1), this can hang the whole system if there's no external timeout to reset it.

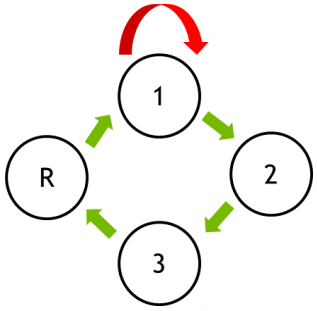

Figure 2. Deadlock in a four state FSM

B. Livelock

A livelock is like deadlock with a small difference, where a cyclic sequential logic (for example: FSMs, Counters) or two interacting cyclic logic (FSM state⇔FSM state or FSM⇔counter) keeps moving between non-reset states indefinitely. The diagram below (Fig 3) shows an FSM is oscillating between two non-reset state (State=1 & 2) Which can hang the whole system if there's no external timeout to reset it.

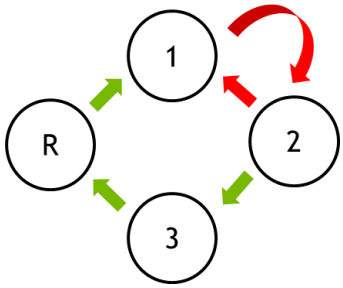

Figure 3. Livelock in a four state FSM

C. Starvation

When a transaction/request/interrupt gets blocked by a higher priority transaction indefinitely, a Starvation can happen, which could result in a system wide hang. For instance, in the diagram below (Fig 4), req1 may cause req2 to experience starvation at the arbiter if req2 is not held stable until a grant is received, or if req1 becomes stuck or caught in a feedback loop.

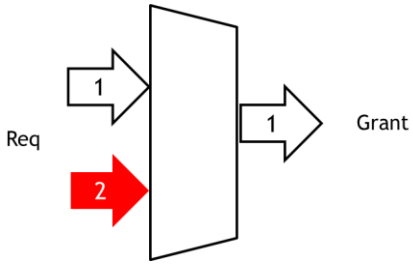

Figure 4. Starvation in two req arbiter

III. FORWARD PROGRESS – HOW TO TEST

There are multiple approaches to performing forward progress checks at the RTL verification level. Historically, simulation-based verification has been our primary approach, with much of the forward progress testing arising as a byproduct of functional testing.

A. Simulation based verification.

Let's have a brief look at some of those approaches.

1. **Corner Case Tests**: Corner case tests target extreme or boundary conditions that may not be covered in normal operation. They test the design's behavior in scenarios where inputs are at their minimum or maximum values, or when specific conditions occur.

2. **Random Tests**: Random tests involve generating input stimuli randomly to exercise a wide range of scenarios. They can help discover unexpected behaviors or corner cases that may not be considered in other test cases. This is limited by constraints on the stimuli.

3. **Stress Tests**: Stress tests push the design beyond its normal operating conditions to identify its limits. They involve applying heavy workloads, high-frequency inputs, or stressful scenarios to test the design's robustness, error recovery mechanisms, and stability.

4. **Error Injection Tests**: Error injection tests intentionally introduce errors or faults into the design to assess its error detection and error handling capabilities. These tests help evaluate the design's resilience and its ability to recover from errors gracefully.

Some of the above, like Stress Tests & Error Injection Tests are intentional forward progress testing in Simulation, while Corner Case & Random Testing are intentionally testing functionality but tends to find forward progress related bugs as well.

B. Formal Verification

The second approach is using Formal verification, which has its additional advantages over simulation as shown in the table below (Table 1).

TABLE I
SIMULATION VS FORMAL

Method	Advantage	Disadvantage
Simulation Verification	Can identify potential deadlocks and livelocks	Cannot prove that the design is deadlock-free
Formal Verification	If proven, can guarantee forward progress	Hard to get conclusive answers

Formal verification is the most rigorous and reliable way to implement forward progress checks and prove the design is hang free. Formal verification relies on specifying and verifying properties to ensure the correctness of a design. These properties serve as formal statements that describe the expected behavior or constraints of a design under testing. Formal engines classify Assertion properties into two main types:

1. **Safety assertion**: These are the properties which are used to formally prove that the expected behavior happens in a finite amount of time. The example below is of a safety assertion where if event 'A' happens, then after 'N' clock cycles event 'B' must happen.

```
assert property (A |-> ##N B)
```

   For forward progress checking, these assertions mostly define the cases where hang issues may happen.

2. **Liveness assertion**: These are the properties which are used to formally prove that eventually something good happens. The example below is of a liveness assertion, where if event ‘A’ happens, then eventually event ‘B’ happens.

```
assert property (A |-> ##[0:$] B )
```

The above assertion doesn’t really benefit the forward progress checking because it’s a weak liveness assertion, and it would prove even if B doesn’t happen. We should change the syntax below to make the assertion effective.

```
assert property (A |-> strong(##[0:$] B ))
            or
assert property (A |-> s_eventually B ))
```

This means that the above assertion would only pass when there’s no trace where event ‘B’ doesn’t happen after event ‘A’.

Of course, the age-old question in every Formal verification engineer’s mind is which type of assertion should I use for my design. Both assertion types have their merits and demerits in the context of forward progress verification,

1. Writing safety assertions for forward progress checking would require a lot more in-depth knowledge of the design, compared to Liveness. Since you are not defining the eventual expected behavior, you need to define the situations where hangs may happen.
2. The Cone of Influence of a liveness assertion is much wider than safety assertion, which means fewer liveness assertions needed compared to safety assertions for the same check.
3. The above comes with a downside of difficulty to get convergence in Liveness assertions vs Safety.
4. Bounded liveness assertions are of no merit, while bounded analysis of safety assertions can still help.
5. Only way forward is to get convergence on a liveness assertion, which would mean heavy abstraction and black boxing compared to safety assertions.
6. There’s an added benefit of testing Safety assertions in Simulation, which isn’t the case with Liveness as it often results in memory blow-out.

The table below summarizes the above discussed differences.

TABLE II
SAFETY VS LIVENESS

Key points	Safety Assertion	Liveness Assertion
Assertion complexity	Complex to write	Simpler to write
Scope	Narrow Scope	Wider Scope
Convergence	Relatively faster	Slower Convergence
Bounded Analysis	Yes	No
Abstraction	Needed for better convergence	Aggressive abstraction needed
Simulation	Can be enabled	Shouldn’t be enabled

IV. FORWARD PROGRESS – FORMAL APPROACH

Let’s discuss the types of checks in detail for both Safety and Liveness assertions.

A. Safety Assertions

Safety assertions with all their merits, aren't the best at defining the expected forward progress behavior, so mostly one would have to try and identify code/design behavior where a hang is possible and write assertions to make sure that doesn't happen. here are few examples.

1. Check that FIFO overflow and underflow doesn't happen, because this can result in corrupting the data and potentially hanging the system, especially if it's carrying any flow control data. The assertions below are examples of fifo overflow and underflow assertions. The timing and signals would vary based on the FIFO implementation.

```
assert property (fifo_full ==1 |-> ~write_done)
assert property (fifo_empty==1 |-> ~read_done)
```

2. Check for State machine never entering an invalid state, else the system can hang, because it wouldn't know where to go from that state.

3. Check that timeout counter is reset after hitting the timeout value only after related logic is triggered.

```
assert property (Timeout_has_happened &&(state==reset_state)|=> timeout_reset)
```

4. Check for transaction ordering, a hang can happen when the design isn't written to handle out of order transactions.

```
assert property (txn_b_started |-> txn_a_finished)
// transaction b should start after a finish.
```

5. Check for Protocol violations, a protocol violation can put receiver logic in unexpected state which could result in a hang. Few examples:

    a. Invalid bus transaction.

```
assert property (~(bus_req && !(bus_read || bus_write)))
```

    b. Improper I2C communication

```
assert property (i2c_ack_expected && i2c_ack_received) //missing ack bit
```

    c. AXI bus protocol deviation

```
assert property (axi_awburst != BURST_UNSUPPORTED) //unsupported burst type
```

6. Checks using counters, for testing input transaction, A should eventually be seen at output B.
    a. if it takes min M & max N clock cycles & if we can keep track of stalls in the system
    b. A safety assertion can be used to prove B should happen Within N cycles of A, if there's no system stalls
    c. Have a counter to start counting when A happens and count till M clock cycles. Then a "flag_bit" is set.
    d. The example below is very simplistic version of proposition above.

```
assert property (flag_bit && ~system_stall |-> ##[0:N-M] B)
```

B. Liveness Assertions

For cyclic logic, like FSMs and counter, the expected behavior is that a counter/FSM should be able to come back to its initial state (in most case it's the reset state, but not always). A simple liveness assertion shown below can be used to model this expectation.

```
assert (State == ~(initial_state) |-> strong( ##[1:$]  State == initial_state))
assert (Counter == ~(reset_value) |-> strong( ##[1:$]  Counter == reset_state))
```

In majority of the designs there are control registers which should also go back to its initial value when the FSM goes back to initial value. The assertion below can be used for not only additional forward progress check but also to check design guidelines.

```
assert (control_register!=init_value) |-> strong(##[1:$]control_register== init_value)
```

when dealing with Starvation points, such as an arbiter, credit-valid or request-valid blocks, the below liveness assertion can be used to define the behavior.

```
assert (req == x |-> strong(##[0:$] grant == x))
assert (credit ==0 |-> strong(##[0:$] reload_credit))
```

While the assertion definition is easy and straightforward, there are multiple challenges when doing forward progress check using liveness assertions. Some of these challenges are shared with safety assertions and some are unique to liveness assertions. Here are the top five challenges,

1. **Convergence**
   Convergence is in general a challenge in formal, but it's a much larger challenge for liveness assertions. Even a simple 5 state FSM could have a huge COI (Cone of Influence) and formal engines could struggle to get a full proof.
2. **Complexity Reduction**
   Finding the right balance of reducing the COI without losing important design elements, and under constraining the design.
3. **Getting Constraints Right**
   Ideally, forward progress is proven in the absence of most constraints, still a minimal guaranteed set of constraints are needed. It is difficult to figure out the minimum constraints to get a proof.
4. **Proof Maintenance**
   Liveness assertions impact a wider scope of the design, so a small change in design can turn a proven assertion into non-convergent. A new set of abstractions and constraints may be required to get a proof again.
5. **Computational Resource Management**
   Formal is resource intensive, especially for big designs. Liveness assertions would require a lot more resources for a lot longer time. Some proofs can take 48 to 96 hours of cpu runtime.

V. Forward Progress – Dos & Don'ts

A. Techniques (To Do)

Here we'll talk about the various techniques which one can use to address the challenges discussed above and how to get full proofs. Most of these techniques are applicable for both safety & liveness, but since Liveness generally is more challenging to converge on, you'll be using these more often with them.

Abstraction – In simple words, abstract everything and anything you can abstract, different types of abstraction techniques help with different convergence issues,

1. **Memories/FIFO abstraction**: Abstracting RAMs/FIFOs etc. helps in hitting the corner case fasters, since one wouldn't have to fill/empty these up to hit those corner case where hang issues might be hiding.

2. **Counter abstraction**: Counter abstraction is key to hit cases where something interesting happens, it also helps in jumping states easily and is quite critical when dealing with slower protocols.
3. **Reset abstraction**: Reset abstraction can come in handy when dealing with huge FSMs, because it'll help in hitting the deep states with tool's proof depth limits.

Black Boxing & Driver Snipping – Black Boxing (Removing a module definition) is quite critical in getting convergence because it removes unrelated logic which could be adding to the complexity of the proof. If one doesn't want to remove the whole block, driver snipping is another way to go, where selected ports can be snipped (equals to removing the whole driver logic).

If the output ports of black boxed module or snipped ports are left open it would result in under-constraining the design. But if those ports are driven by a constant value (stuck at 0 or 1) that could result in over constrained design.

It would be an additional effort to model the driving logic to get the exact constraints. It also comes at the expense of some added complexity.

If getting the exact constraint is time intensive, it's okay to under constraining the design.

Assume-Guarantee – When dealing with a non-convergent assertion, the user can add helper assertions which once proven, can be used as assumption to make COI smaller and get convergence on the primary assertion.

```
assert primary (seq_1 |-> strong (##[1:$] seq_N))
                    Into
assume helper1 (seq_1 |-> strong (##[1:$] seq_2))   //property proven before
assume helper2 (seq_2|-> strong (##[1:$] seq_4))    //property proven before
assert primary (seq_1 |-> strong (##[1:$] seq_N))
```

In the above example, assertion "primary" was not converging, but when two helper assertions "helper1 & helper2" were proven and then used as an assumption, it reduced the complexity of primary assertion to have it converge.

Case Splitting- This technique is used to reduce one assertion with high complexity into multiple smaller assertions, especially works well in Liveness assertions since FSMs are easy to case split.

```
assert (state == ~(reset_state) |-> strong(##[1:$] state == reset_state))
                    Into
assert (state == state_1 |-> strong(##[1:$] state == reset_state))
assert (State == state_2 |-> strong(##[1:$] state == reset_state))
```

Taking above code as an example, the assertion checking "state" must eventually go back to rest from a non-reset state, can be case split into "state" must go back from each of its non-reset state to reset. If a subset of these assertions does not converge, one can further case-split it to check transition between each valid state.

B. Gotchas (Don'ts)

There are few gotchas one need to keep in mind to avoid getting bamboozled later by false passes and/or missed bugs.

1. Don't use weak liveness assertions for forward progress testing, it's sometimes easy to miss the keyword "strong". Make sure you scan all your liveness assertions for 'strong' or 's_' if using eventually keyword.
2. There's no bounded proven in Liveness assertions, just treat all your bounded proven assertions as failures.
3. Formal regressions are important, a proven assertion now, might become bounded in the future and bounded liveness assertions are only as good as a failing assertions.
4. For faster convergence of a liveness assertions reducing complexity helps much more than adding more computational resources.

5. Avoid mixing liveness assertions and safety assertions in one run, as it makes it harder for tool to get convergence.

VII. Case Study

We did formal verification based forward progress testing on a few different designs using both safety and liveness assertions. These designs were a mix of FSMs, FIFOs, Arbiters, and other glue logic. All these designs were all data transportation modules and had varying sizes from small sub-module to a large block. The run time limit and compute resource allocation were kept same for both types of checks.

We employed three types of formal flow for forward progress testing.

1. **Forward Progress Liveness**
    a. Each test block had one formal setup with all liveness assertions.
    b. Minimal constraints only
    c. Both safety and liveness constraints allowed.
2. **Forward Progress Safety**
    a. Each test block had one formal setup with all safety forward progress assertions.
    b. All necessary constraints only (more constrained than liveness)
    c. No liveness constraints, only safety constraints.
3. **Forward Progress No-timeout**
    a. All the design blocks with timeout logics had formal setup.
    b. Only enabled assertions with functionality testing (not forward progress)
    c. All timeout counters were cut.
    d. All necessary constraint

Below pie chart (Fig 5) represents the forward progress bug distribution by the type of formal flow, from the sample set of runs.

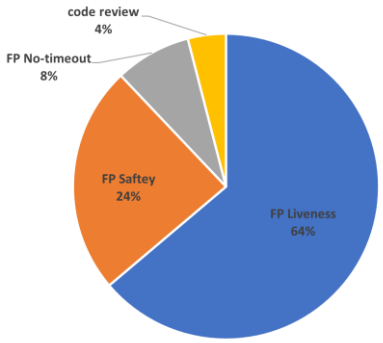

Figure 5. Percentage of forward progress bugs by formal flows

Undoubtedly, liveness assertions proved most effective in detecting forward progress bugs. However, achieving complete convergence for all liveness assertions posed a significant challenge. The graph below (Fig 6) illustrates the incremental assertion convergence achieved with each additional technique discussed in Section V.

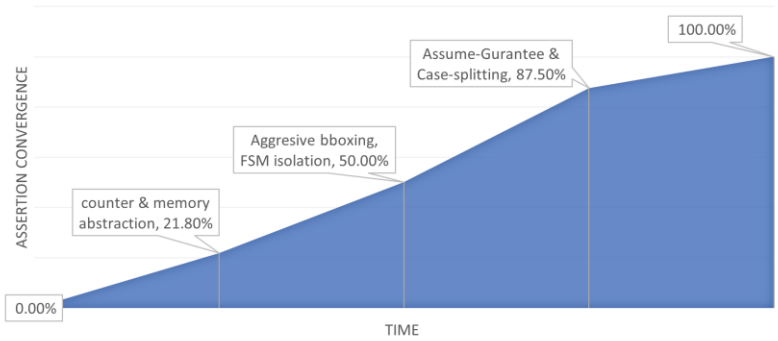

Figure 6. Progress in assertion convergence incrementally using various convergence techniques.

The box with description in the above graph indicates when the specific technique was added to the flow and what percentage of assertion were converging till then. We signed off on the design after hitting 100% convergence of liveness assertions and hitting sufficient bounds (with bounded analysis) of safety assertions.

Here's the final summary of us comparing liveness assertions vs safety assertions.

TABLE III
SAFETY VS LIVENESS RESULTS

Key points	Safety Assertion	Liveness Assertion	Summary
Bring-up		WINNER	Liveness assertions were simple to write, with minimal constraints
Convergence	WINNER		faster to converge on all Safety assertions
Bug Count		WINNER	Forward progress bugs found by Liveness assertions was much higher
Coverage		WINNER	Covered more design, because of wider scope
Signoff	WINNER		Faster to signoff for Safety, considering bounded analysis
Verification Confidence		WINNER	Higher with proven Liveness assertion because it proves the intent.
Overall		WINNER	Nothing beats finding most bugs

VIII. CONCLUSION

We discovered that while liveness assertions could take lot of time and effort to converge, initial failures weren't hard to hit. In fact, we found many early bugs with liveness assertions even when safety based forward progress setups weren't up and running. Forward progress bugs found using Liveness Assertion were easily 4 times more than the safety assertions.

Safety assertions had the benefit of being easier to converge and the ability to run in simulation, which made it easier to sign off on safety check. But confidence to sign off on the design only comes with proven liveness assertions as it's guaranteeing a hang free design component within the constraint.

Often formal verification engineers dismiss liveness assertions focusing only on safety assertions, which does have merit, especially in functional testing. But when looking at forward progress testing, Liveness assertions outshines Safety assertions in most places.

The above finding is based on our experience, it could vary by designs. I would recommend having a healthy mix of both, not discounting either in exchange of the other.

REFERENCES

[1] M. Munishwar, N. Zaman, A. Jain, H. Singh, V. Singhal, "Architectural Formal Verification of System-Level Deadlocks", DVCon 2018
[2] Book, D. Perry, H. Foster, "Applied formal verification: for digital circuit design", McGraw Hill, 2005
[3] Book, Formal Verification Erik Seligman, Tom Schubert, and M V A. Kiran Kumar, "An Essential Toolkit for VLSI Design", 2015.
[4] IEEE Std 1800™-2017, IEEE Standard of SystemVerilog – Unified Hardware Design, Specification, and Verification Language.
[5] J. R. Maas, N. Regmi, A. Kulkarni, K. Palaniswami, "End to End Formal Verification Strategies for IP Verification", DVCon 2017
[6] Adnan Aziz , Vigyan Singhal , Robert K. Brayton, Verifying Interacting Finite State Machines : Complexity Issues, 1993
[7] Mark Eslinger, Jeremy Levitt, Joe Hupcey, "Deadlock Verification for Dummies – Easy Way of Using SVA & Formal", DVCon US 2020
[8] Book, Pallab Dasgupta, "A Roadmap for Formal Property Verification", 2006