2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Sparking UVM stimulus via state design pattern

Debarati Banerjee
Google IT Services India Private Limited,
Bangalore, India
bdebarati@google.com

Nikhil Singla
Google IT Services India Private Limited,
Bangalore, India
nikhilsingla@google.com

Rohit Jindal
Google IT Services India Private Limited,
Bangalore, India
rohitjindal@google.com

Abstract - ***With the silicon industry moving towards zero post silicon bugs, the significance of extensive verification at pre-silicon level is paramount.One of the major challenges in pre-Silicon verification is to develop a test suite strong enough to cover all the corner cases unearthing the potential bugs. Depending on the complexity of the design, adding complicated test scenarios manually is tedious and at times difficult to maintain. As the DV cycle continues, more and more UVM sequences are coded and many conditional paths get added in the testbench due to inter sequence dependencies. Any further change needs thorough understanding of all the sequences and requires modifications at multiple places. In this paper, we propose a smarter state pattern based UVM sequence library and methodology that manages all the scenarios efficiently as the states are kept aligned with the DUT system state.***

I. Introduction

Any testbench scenario can fundamentally be broken down into a stimulus which targets the DUT in one state and on completion of activity moves the DUT to the same or different state. Many such stimuli are cascaded to come up with complex scenarios. Conventionally UVM testbenches have multiple stimuli (sequences) which are run in specific orders to obtain the required scenario. The problem with this approach is that there is a lot of interdependency of sequences and the user has to ensure every test, virtual sequence is following this. For instance,

- Clocks sequence can't be run before power sequence
- Data sequences can't be run before clock and reset sequences
- Multiple power up/down dependency due to multiple power/clock domains

Managing these dependencies during the initial phase is still manageable but when new dependencies need to be added due to new scenario coding, struggle starts to decode the sequences and usually end up breaking existing test cases.

The methodology proposed in this paper is a smart way of managing the sequences via a state pattern based sequence library. The design can be logically split into different units or states. The concept of state patterns is leveraged to layer different verification sequences. The states maintained in the testbench handles the DUT states dependencies hasslefree. Any new additions and modifications is just adding the new sequence to the required state.

The full test suite can be visualized as a combination of states where every state has a bunch of sequences associated with it. Each sequence will have the initial state as one of the valid design states and will lead to another valid design state. The test algorithm picks a random sequence for the initial state. This sequence can either lead to a different state or loop back to the same state on completion. Once the sequence is complete, algorithm picks random sequence as per the new state .This loop continues 'n' number of times to cover 100% cross scenario. Fig. 1 is the pictorial representation of the different transitions possible for 2 states in the proposed methodology.

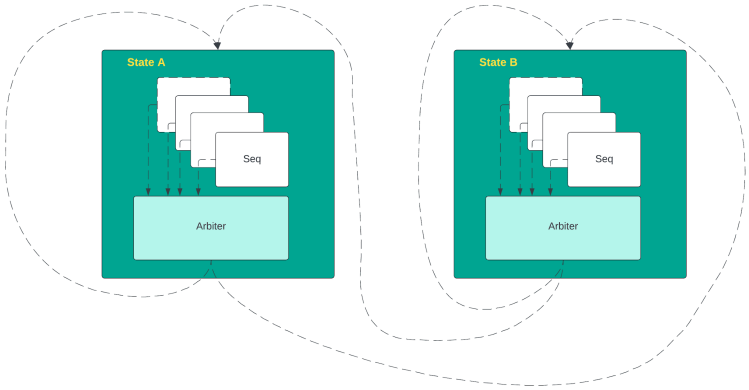

Figure 1 : Different possible transitions for 2 states in the proposed methodology.

The main advantage of this algorithm is that there is no need to code 100s of test cases for multiple scenarios, once the sequences are correctly associated with the states, everything is automatically handled and cross scenarios can be achieved very quickly, similar to SV constraints. The solution can be implemented with just 1 sequence in each state and as more and more scenarios are coded, user just need to add the sequence to the corresponding state sequence library.

The proposed methodology involves :

- State aware sequence library
  - Data structure to store the sequences and the end states.
  - It contains different APIs to add/remove/skip required sequences for any valid design state.
  - Biasing hooks to modify the selection algorithm on the basis of past runs.
- Test
  - Top level test, executing the sequence libraries in accordance to state encountered
  - Control to replay one particular arc
- TB tool
  - For complex designs, the number of possible states to be verified is high. For those scenarios there is also a provision to add sequence and states in XLS format. TB tool which is provided as a part of this dumps the UVM data structure which can directly be plugged into the setup.
- Directed scenarios command line
  - User can pass the list of sequences in order of execution. Top test will pick only the given arc. This is helpful to create deterministic scenarios

The whole scheme is analogous to linked list, where each node of linked list is a system state of DUT, and houses multiple sequences. Each sequence executed will give the next state of DUT and so on. Modifying the scenarios is very straightforward. Like a linked list, nodes can be removed, added, sandwiched without any change in base code.

One added advantage users get is the logical boundary for the checker's implementation. Say after reset, all the signals should be parked to reset value and once DUT is in powered down state, all outputs will be isolated. These checks can easily be implemented in the respective states, the user is required to call the checker before exiting the state.

II. Conventional Approach vs Proposed Methodology

Figure 2 : Steps of sequence to verify pst transitions using conventional approach

Let's consider a design with 2 power domains such that out of reset PD1 is powered up and PD2 is a sub power domain of PD1. The conventional way of verifying the pst transitions of this system would be to write a uvm sequence and invoke all the power up/dwn sequence in sequential fashion. Fig 2. depicts this approach.

For the proposed methodology, we need to break it into multiple logical states. One way of inferring logical states would be based on the power domain.The logical states for such a design could be :

1. State1 (A) : only PD1 powered up(i.e. PD2 is OFF)
2. State2 (B) : Both PD1 and PD2 are powered on
3. State3 (C) : Full system powered down i.e. Both PD1 and PD2 are OFF

Each state can be mapped to a sequence library which contains all the valid sequences which could be run in that state. In the current example, there will be 3 such sequence libraries. Each sequence library will have the required power up/down sequences. Fig3. represents the state diagram for the same.

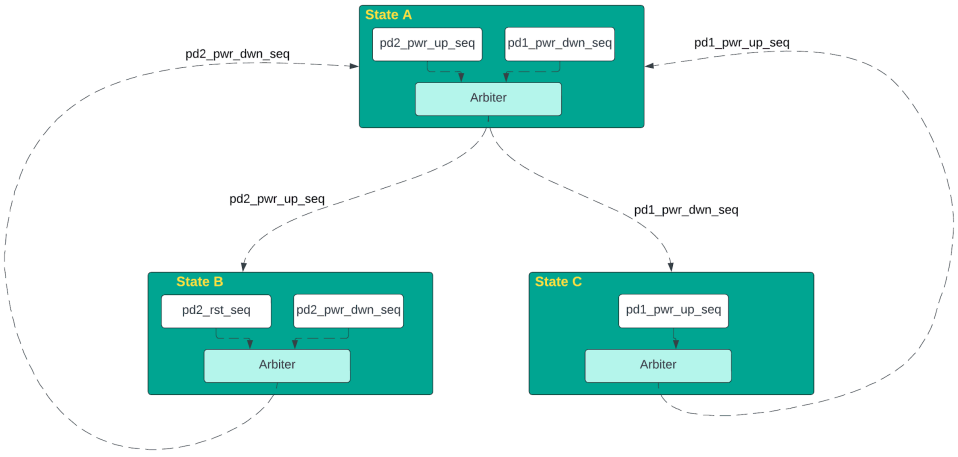

Figure 3 : State Diagram for the proposed approach

The test algorithm picks a random sequence for the initial state. Let's say the initial state is "state C". In the current example, state C has only 1 sequence, so the test algorithm executes pd1_pwr_up_seq and the current state moves to state A. For state A, the test algorithm can pick 1 of the 2 sequences i.e. either execute pd1_pwr_dwn_seq and move back to state C or execute pd2_pwr_up_seq and move to state B. For the case when the algorithm picks the 2nd option, the current state moves to B. For state B, there are 2 sequences. One sequence(pd2_rst_seq) asserts reset for PD2 domain and the other sequence(pd2_pwr_dwn_seq) powers down the PD2 domain. For the iteration when the arbiter executes pd2_pwr_dwn_seq and current state moves back to A. In this scheme we can constrain the randomisation to prioritize picking of the untraversed path. So now, the algorithm executes pd1_pwr_dwn_seq.

Now suppose we want to run some functional scenarios with PD2 ON and also keep the existing variant of no traffic enabled. In the conventional approach, the sequence needs to be updated to incorporate the sequence where as in the proposed methodology, we just need to add the functional sequence to the state2 i.e. B.

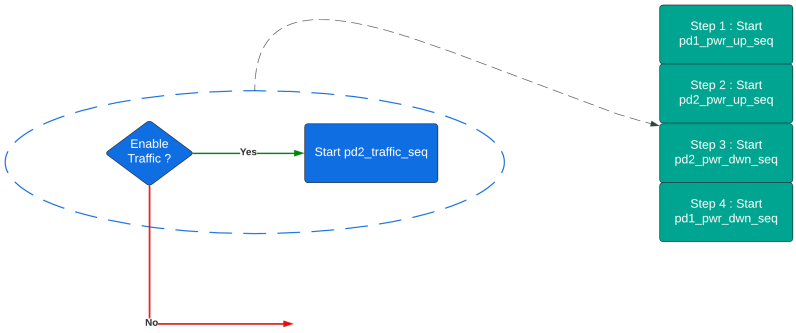

Figure 4 : Conventional Approach

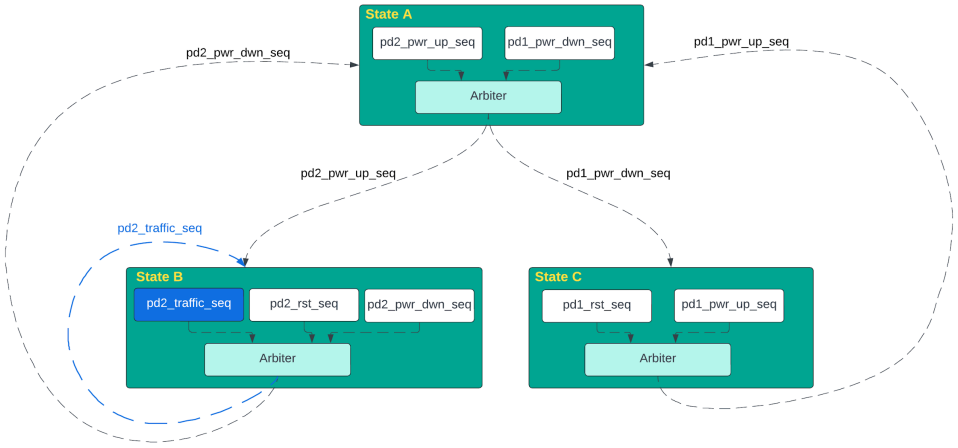

Figure 5 : Proposed Approach

Fig 4. depicts the updated flow for the conventional approach and Fig 5. depicts the state diagram for the proposed methodology. In the proposed scheme, all we need to do is add the functional sequence to the sequence library of state B and then the rest is taken care of by the test algorithm. Thus, with the proposed methodology both the scenarios i.e. with and without traffic is created with minimal manual overhead since APIs are provided to add/remove sequences from a state

III. Application

The proposed methodology makes it easier to create multiple complex scenarios with minimal manual efforts. Consider a scenario where there is a setup to run 2 sequences : seq1, seq2(corresponding states being state1 and state2 respectively). Fig 6. demonstrates the state diagram for the same. As explained previously, each state in this context is a sequence library.

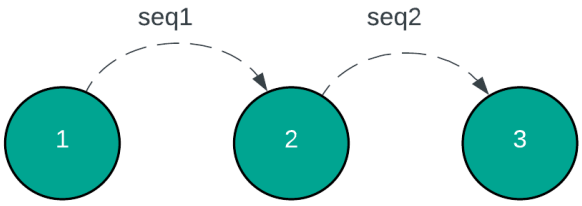

Figure 6 : State diagram showing 3 states in the proposed methodology

Now, if there is a requirement to execute a 3rd sequence(seq1.5) between seq1 and seq2. Just like linked lists, now a node needs to be added between state1 and state 2 i.e. we need to create a state between state1 and state2(say state1.5), add seq1.5 to it, assign the next state of this sequence as state2 and update the next state of seq1 to state1.5 from state2. Thus the entire task of manually adding the new sequence in between 2 existing sequences is reduced to 1-2 API calls which in turn enhances code readability and reduces code maintenance overhead. Fig 7. depicts the state diagram after inserting a new node.

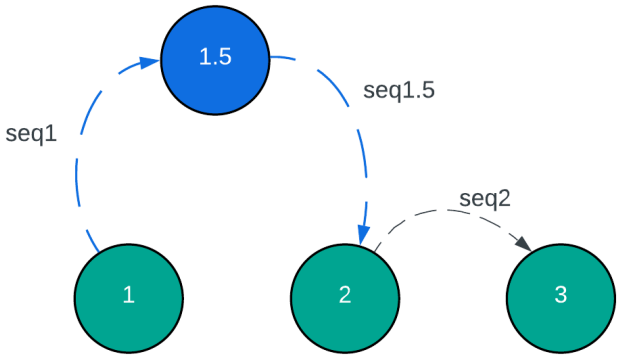

Figure 7 : State diagram showing addition of an intermediate in the proposed methodology

The generic nature of this solution makes it applicable to both simple and complex systems. Due to its portable and compact nature the same setup at IP/subchip level can be easily extended to the SOC level(if the sequences are portable)

IV. Related Work

The usual approach has been to code separate sequences for every scenario which not only increases the code length but also makes it difficult to maintain across multiple projects. Any new verification requirement often causes a lot of DV sequence changes and leads to some redundant code.

UVM provides the sequence library base classes but without the state knowledge it's just a database of sequences. The concept explained in this paper is organizing the test stimulus in a structured way using a state pattern which once implemented handles the complex corner cases hasslefree.

V. Results

The idea is proven on multiple subchips. It has helped multiple DV teams achieve a good amount of functional coverage much early in the project cycle. Due to its generic and portable nature, any combinations of the existing/coded sequences can be executed right from the start itself thus enabling testing of a lot of corner case scenarios quite early in the cycle.

VI. Conclusion

The proposed solution significantly reduces the manual overhead of creating multiple cross functional scenarios. High flexibility and scalability of this solution enables it to be deployed for all kinds of design. Code maintenance overhead is reduced significantly. It can be easily ported across IP, subchip and SOC TB. It increases the overall productivity and efficiency of the team significantly. Less manual overhead and faster TB infra deployment enables DV folks to spend more time on genuine design issues thus giving higher confidence before tapeout.