2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

A Graph-Based UVM Generation Framework for Complex State Machine Verification

Application to PCIe Link Training

François CERISIER, CEO, AEDVICES, Grenoble, France (francois.cerisier@aedvices.com)

Philippe LEDENT, Research Engineer, AEDVICES, Grenoble, France (philippe.ledent@aedvices.com)

Eric HARGOUS, CTO Verification, AEDVICES, Sweden (*eric.hargous@aedvices.com*)

***Abstract*—The verification of complex designs which include big control and protocol state-machines has always been a challenge. Although SystemVerilog and UVM have brought to the verification community the ability to create constrained random scenarios by means of test sequences and sequence libraries, the management of complex protocols from those sequences often leads to spaghetti code, hard to maintain sequence libraries, directed tests or worse, not verifying all the targeted features. In parallel, we have seen the emergence of graph-based approaches resulting in the development of the PSS (Portable Stimulus Standard), bringing the capacity to think differently about the use case scenarios, allowing them to be combined in a much bigger picture and enabling cross platform reuse. Those techniques are efficient at System Level and for verification of complex SoC. However, using them requires learning a new language, integrating new tools and are therefore less of an added value in complex IPs and subsystem projects already using UVM.**

Meanwhile, SystemVerilog provides interesting language constructs that can leverage standard UVM sequences for a more efficient constrained random generation, therefore accelerating the coverage closure of complex state machine designs. On the generation side, the `randsequence` keyword is powerful in generating graph-based scenarios providing that we stick to a well-structured template. Using its full capabilities and adding a few tricks we can even enable fully controlled graph explorations as well as automatic coverage closure completion, whether the design is fully controlled by the testbench or the testbench reacts to the design behavior. On the coverage side, SystemVerilog covergroups allow the creation of state and state transition coverage. It is also possible to query the covergroups to check the completion of the scenarios. Integrated into a UVM sequence, we can define an automatic coverage closure graph-based scenario which will automatically cover all the required states and transitions of the design under test.

This paper presents the overall approach, proposes an application proven template for the `randsequence` graph-based UVM sequence and leverages this by automating the template generation. A concrete example, based on the PCIe link training state-machine is then presented.

The generation script and case example base class will be later provided as a gitlab project link and an online generation tool available on www.aedvices.com/gbug (Graph Based UVM Generation)

Keywords—UVM complex sequences; coverage closure; graph-based verification; automatic generation

I. INTRODUCTION

The verification of designs containing complex state-machines requires building test sequences of a level of complexity often overpassing the complexity of the initial designs. But verifying such complex state machines does not have to be complex. For instance, designs such as Power Management Controllers, Cache Coherency

Interconnects, Complex DMAs, Pulse Width Modulators, or PCIe Root Complex interfaces, have in common that they rely on complex state-machines (see Figure 1 Full PCIe Training FSM, including sub-states as an example).

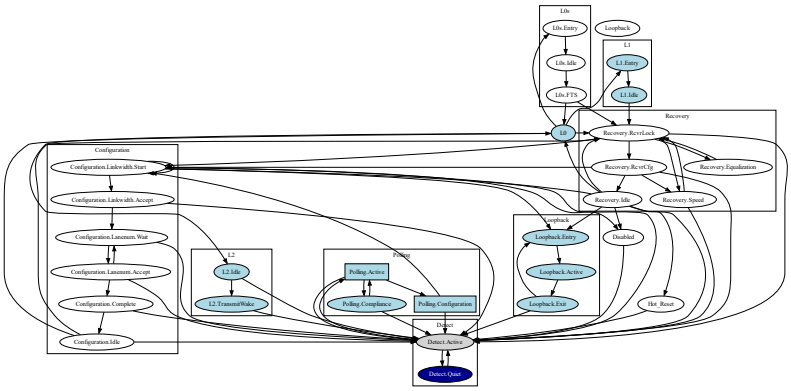

Figure 1 Full PCIe Training FSM, including sub-states

These state-machines look complex at first glance, and it is quite tempting to reproduce test sequences that are just implementing all these possible transitions. However, looking locally at the most complex node (in this case Recovery.Idle), we end up having to manage only 7 possible transitions (see Figure 2 Recovery.Idle state and its possible transitions). Writing a single test sequence that has for prerequisites to be in Recovery.Idle then becomes "quite easily" manageable.

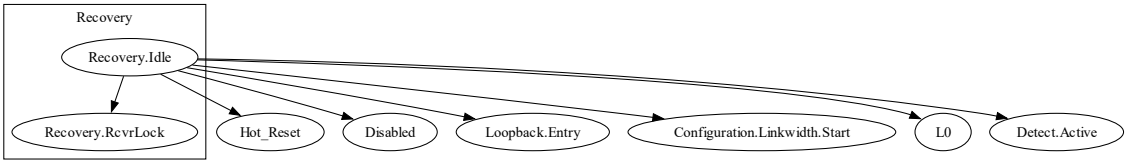

Figure 2 Recovery.Idle state and its possible transitions

However, the local aspect of such simpler sequences masks the overall state-machine from the implementer, leading to code harder to maintain and ultimately a coverage closure harder to reach. Randomly covering the state-machine or demonstrating that the set of sequences implements the full state-machine becomes challenging. The use of the SystemVerilog `randsequence` keyword can palliate this, easing the overall state-machine definition, while keeping local implementation of individual choices. In Figure 3, we use the `randsequence` statement to generate a sequence of actions, starting from `Polling`, going to `Configuration` and then to either `L0`, `Recovery` or `Detect`, each time calling an action plan before moving to the next production statement.

```
        task body();
            randsequence ()
                Detect:        { do_something(); } Polling ;
                Polling:       { do_something(); } Configuration    ;
                Configuration: { do_something(); } L0           |
                               { do_something(); } Recovery     |
                               { do_something(); } Detect     ;
                //...
            endsequence
    endtask
```

Figure 3 Simple randsequence production example

This kind of randsequence production is effective to generate purely random state-machines where transition actions should be followed depending on the state to reach, and when the transition can be selected randomly with no possible reactions to the design behavior. In this paper, we showcase how to use the randsequence statement in a pragmatic, yet effective way, leveraging it in a fully graph-based verification approach, dealing with both fully random and reactive choices. This is demonstrated using the PCI-Express (PCIe) training link state-machine of the datalink layer.

As the state-machine diagrams are well defined by their transitions, it should also be possible to automate the generation of a sequence. Such generation will improve productivity as well as correctness of the sequences. It will

be possible to review the source definition (a graph) and the generated code easily by direct comparison with the FSM specifications.

The remainder of the paper is organized as follows: Section II further illustrates the limitations of solely relying on randsequence for test generation and other similar approaches. Section III presents how randsequence can be leveraged as a backbone for graph-based verification. Section IV presents our graph-based solution for randomized test generation. Section V details the automated test generation. Section VI showcases a reproducible concrete use-case built around the PCIe Training sequence from Figure 1 for which we analyze the results, Section VII VII concludes.

II. EXISTING APPROACHES AND THEIR LIMITATIONS

A. Ad-Hoc UVM Patterns

A state-of-the-art UVM testbench implements the verification environment using Verification IPs, interfaces, assertions and scoreboards. Figure 4 illustrates how they are usually connected. For designs with complex state-machines, one needs to develop the right set of test scenarios in the form of layered constrained random sequences, as well as the *checkers* and the *coverpoints* to ensure that all features are properly exercised.

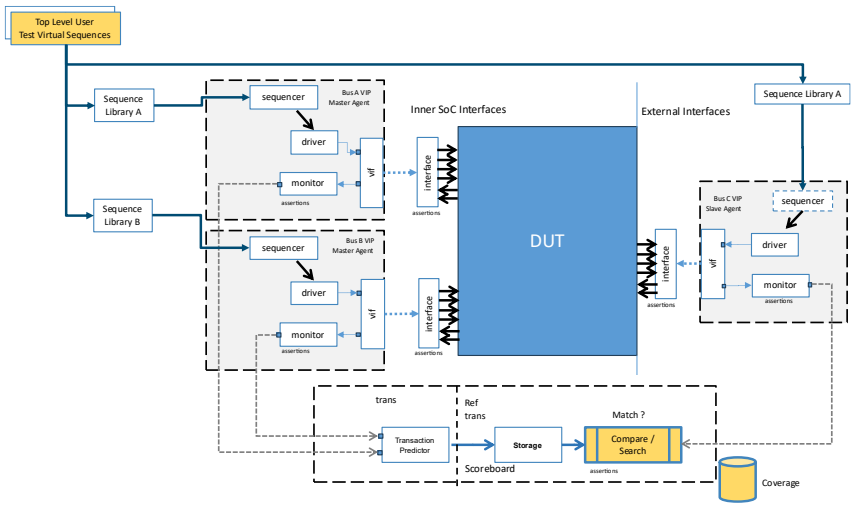

Figure 4 - Standard UVM Environment

However, when it comes to modeling and covering state-machines in a verification environment, very few applicable patterns are proposed. The methodology of [2] provides an interesting way to model a state-machine using a software design pattern. This pattern is however more applicable to FSM modeling based on direct inputs than crawling randomly through the transitions of a FSM. This could be easily hacked using random variables instead of direct control inputs.

```
class FSMExample;
  local fsm_t currentState;
  function fsm_t calculateNewState(fsm_t state, inputs);
     fsm_t result;
     case ( state )
        S0 : std::randomize(result) with { inside { S0, S3, S10 } };
        S1 : std::randomize(result) with { inside { S0, S4, S9  } };
     endcase
     return result;
  endfunction

  // Main action function
  function void doAction(Input inputs);
    case (currentState)

      fsm_reset: begin
        doActionForState_reset(inputs); // or call a UVM sequence
        currentState = calculateNewState(currentState, inputs);
      end

      fsm_init: begin
```

```
        doActionForState_init(inputs);
        currentState = calculateNewState(currentState, inputs);
      end

      // Add more state handlers here...

    endcase
  endfunction

  task body();
     while ( ! complete ) begin
         doAction(inputs);
     end
  endtask
endclass
```

Alongside, the online Verification Academy page [3] proposes a "Unified" approach to verify complex FSM based on tasks calling each other through random variables and leads to a quite similar approach.

These approaches are mostly based on modeling the FSM, generating random variables and exercising the associated action based on the generated variables. They require explicit coding of each state and transition, leading to a weak maintanability in the long term. Additionnaly, [3] uses recursive calls, leading to stacking task calls, which could be an issue in a big FSM.

B. UVM Sequence Stack and randcase

Another quite common method is to define atomic sequences for each individual edge, or, at minimum, for each objective node, for instance:

```
    typedef class idle_seq;           // Idle state of the Tx Engine
    typedef class configure_seq;      // Write to Configuration Registers
    typedef class fill_tx_buffer_seq; // Fill the TX Buffer with data
    typedef class start_tx_seq;                 // Start the TX Engine
    typedef class wait_for_interrupt_seq;       // Wait for an interrupt from
the RX Engine
    typedef class read_rx_status_seq; // Read the RX Status Register
    typedef class read_rx_buffer_seq; // Read the RX Buffer with data
```

Such a sequence structure requires to develop `body()` tasks that make random choices amongst alternative cases, then calling the appropriate sub-sequence of this sequence library.

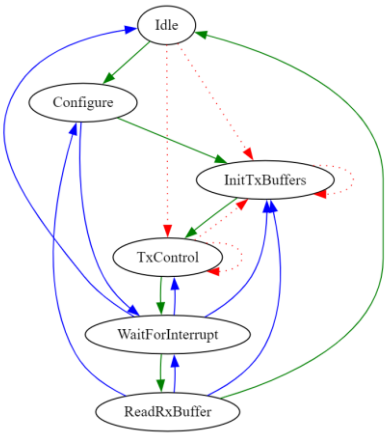

Figure 5 Example of a FSM to cover

For instance, using the following `randcase`:

```
class idle_seq extends uvm_sequence;
  // ...
  rand int unsigned idle_delay;

  rand bit can_go_to_InitTxBuffers;
  rand bit can_go_to_TxControl;
  virtual task body();
    // Be Idle
    #(idle_delay*1ns);
    // Compute can_go_to_InitTxBuffers and can_go_to_TxControl
    // depending on simulation history
    //...
    // Take actions to move on
    randcase
      1                     : `uvm_do ( configure_seq );
      can_go_to_InitTxBuffers : `uvm_do ( fill_tx_buffer_seq );
      can_go_to_TxControl     : `uvm_do ( start_tx_seq );
    endcase
  endtask
```

In such a sequence, the possibility to execute one of the following sub-sequences is made by computing the `randcase` weight depending on the simulation history. This type of sequence, however, faces three issues:

- Sequences are calling each other along the hierarchy of nested sequence calls. One may have to face numerous stacked sequences when maintaining and debugging.
- This pattern does not provide the ability to cope with actions reacting to the design's behavior. This can be partially implemented within each `case` of the `randcase`, by polling or synchronizing to external actions.
- No exit conditions are defined, and sequences will call each other in an infinite loop, growing the stack of nested sequence calls to infinity. To avoid this, one may define a possible exit condition choice in one, any, or all sequences, allowing an exit possibility that still has to be reached.
- Even with an additional exit condition, this would be very poorly controllable as one of the possible choices in the sub-sequence hierarchy. Some tests may exit immediately, while others may run a huge number of sequences.

C. Comparison with PSS

One recent development in the EDA industry concerning test generation is PSS (Portable-test and Stimulus Standard). The main asset of PSS is seeking reusability. It builds upon the observation that testing scenarios are usually the same across various levels of abstractions (IP-level, System-level, etc.) multiple platforms (simulation, FPGA emulation, etc.), and even projects. The verification intent (partial ordering of desired actions) remains constant, but the implementation of each action and the possibility of other actions being inserted in between may differ. One example is the classic "read, modify and read-back" scenario that could fail if other actors are also writing at the same address, and whose implementation differs depending on the levels of abstraction, platform, and project. Having a constant high-level structure gains time.

Despite providing semantic constructions for writing state machines with the "State flow object" the works in [5] and [6] show that the semantics of various PSS constructs are ambiguous by nature and rely on the tool provider's interpretation. The work in [5] illustrates the difficulty of writing a multi-agent model where each agent has its own FSM. It was shown that in PSS, the most easily understandable and controllable modelling technique consists in a single FSM combining the state information of all agents, thus losing modularity when increasing the number of agents because in PSS behavior is governed by constraints which all need to be updated. In [6], the same authors explored the PSS semantics by expressing them in the form of a small FSM for each action (wrapper for executable code) and flow object (mechanism to share data between action) and combining them together using formal methods. They discovered that PSS semantics are ambiguous (in terms of defining an FSM for each flow object) and that PSS was never meant to describe a full system and extract test sequences. Rather, it takes a verification intent (minimalistic and voluntarily incomplete partial ordering of actions) and infers other necessary intermediate actions. The interpretation of these ambiguities to determine the state-space expressed by a PSS model is left to the tool providers. The proposed algorithm in PSS documentation [7] yields the shortest possible test which cannot cover the entire state-space of a desired FSM model. Fully grasping and maintaining a PSS model appears daunting.

Contrary to PSS methodology that requires learning a new language, our approach in this paper remains in the confines of SystemVerilog and UVM, cleverly exploiting the `randsequance` construct which is not yet used at full capacity by the community. In addition, the input to our methodology is graph written as a simple CSV or Excel file with a set of nodes (States) and edges (transitions). The expressed graph is explicitly understandable and is easily controllable.

III. RANDSEQUENCE AS A BACKBONE FOR GRAPH-BASED VERIFICATION

In this section, we explore the SystemVerilog construct `randsequence` available to UVM. We will see how it already is an improvement over nested UVM sequence, and that we can cleverly control it for graph state-space exploration. It is essential to our proposition.

A. *Language Considerations concerning randsequence*

The randsequence keyword is described in the very last chapter of the Constrained Random Value Generation section of the IEEE SystemVerilog reference manual [1] as a random generator for randomly generating structured sequences of stimulus such as instructions or network traffic patterns. Trainings, tutorials, and courses in UVM rarely cover it. Tool support is limited and different subsets are supported by different CAD vendors. It is nevertheless a very interesting language construct to implement random choices, based on a previous path.

Table 1 shows a simple example of a randsequence, where the first production element is INIT which itself is composed of a random choice, with a weight of 2 to select SETUP, and a weight of 1 to select WRITE. SETUP is a sequence production and will execute CONFIGURE followed by WRITE in a sequential order, where CONFIGURE and WRITE are terminal production items that actually execute SystemVerilog code. Here, they call UVM sequences.

```
task body();
  randsequence(INIT);
    INIT     : SETUP:=2 | WRITE:=1;
    SETUP    : CONFIGURE WRITE;
    CONFIGURE: { `uvm_do ( pcie_config_seq); } ;
    WRITE    : { `uvm_do ( pcie_fill_tx_buffer_seq ); } ;
  endsequence
endtask
```

Table 1 - randsequence simple example

Once *terminal nodes* are reached, no production is performed and the randsequence completes.

Other aspects of randsequence include repeat iterations, conditional production (if and case), concurrent production (fork), and the production of data for the next node. However, these are not fully supported by the major CAD vendors and cannot really be used without taking care of portability between tools.

Nevertheless, since the weights can be variables, we can imagine that each choice has a variable weight which is re-evaluated at any time during the randsequence execution so that we gain full control or to the contrary keep a full random state machine execution.

B. *Randsequence Template*

As described in III.A, randsequence can easily be used to define random state-machines. At first, describing the use-case scenario defined in Figure 5 could be defined by Table 2.

Using this template, the choice is not delegated in each nested UVM sub-sequence but is rather consolidated in one top level UVM sequence. This avoids stacking the sequence calls as seen in section II.B.

```
randsequence(Idle);
  Idle:         {`uvm_do(config_seq)  ;} Configure       |
                {`uvm_do(init_tx_seq) ;} InitTxBuffer    |
                {`uvm_do(tx_start_seq);} TxControl       ;
  Configure:    {`uvm_do(init_tx_seq) ;} InitTxBuffer    |
                {`uvm_do(wfi_seq)     ;} WaitForInterrupt;
  InitTxBuffer: {`uvm_do(tx_start_seq);} TxControl       |
                {`uvm_do(init_tx_seq) ;} InitTxBuffer    ;
  TxControl:    {`uvm_do(wfi_seq)     ;} WaitForInterrupt|
                {`uvm_do(tx_start_seq);} TxControl       ;
  //...
endsequence
```

Table 2 Randsequence FSM Template

There are still a few issues with this code. First, since all productions lead to another production, the randsequence will never end. Exit conditions are mandatory.

Second, in this template we mixed, in the same production "Idle", the notion of choices using '|' and the execution blocks. In this simple example this is fine, but on a more complex randsequence , it potentially makes it difficult to understand and may be hard to maintain.

Next, the execution blocks using the "{}" notation are performed when choices are made. They are therefore meant to execute the actions that are responsible for moving from one state to the next state, and not when the state is reached.

Another point is that the randsequence is fully responsible for choosing the next state. This leaves no opportunity to react to the design's behavior, such as from a polling sequence, or dedicated interrupts. Once a choice is made, the design must be in the correct state. But in some cases, the testbench must be reactive to the design,

especially in case of a full separation of concerns between inner and outer functional sequences which live their lives independently.

Finally, although it is great to randomly cover a state-machine, there is no automatic coverage closure, nor coverage at all.

A ***template*** should therefore provide the following features:

1. Integration in the UVM sequence model, as a virtual base sequence that users can inherit in order to implement their own actions on any edge or on any state.
2. A clear distinction between edges and states.
    - We recommend to use edges for code that are meant to instruct the design to the next state
    - We recommend to use states for code that are meant to react to the design and instruct to move (goto) the next state depending on the design feedback.
3. Controlled Weighted choices: Users should be able to define the default weight of the graph decisions, making for instance more weight on normal operations than on error conditions.
4. Exit conditions: when the targeted coverage is reached, or after a maximum number of iterations.
5. Control of the decision:
    - Jump or Goto actions: allowing the users to decide and force a transition, based on reactions to the design.
    - Disable / Enable: any edge at any time, allowing users to make certain choices depending on the execution history or on external events and conditions.
6. Reusable Coverage Model:
    - State and transition coverage should be monitored. This will allow users to ensure their tests are doing what was targeted. However, embedding this transition coverage only measures the "intent" coverage, not the actual monitored transitions from the design behavior as such.
    - The coverage should therefore be reusable and it should be easy for the user to instantiate the same covergroup in their own monitors.
7. Utility Checkers as a set of functions that users can call from their scoreboard to check the transitions.

IV. A POWERFUL GRAPH-BASED UVM SEQUENCE PATTERN

To solve these problems, we defined a `randsequence` pattern which we embed in a scenario oriented UVM sequence.

A. UVM class pattern

In an effort to separate the graph definition from the action implementation, a class pattern is proposed. The `randsequence` pattern is implemented as a `uvm_sequence` and is implemented in the `body()` task. Each **edge** and **state** production of the `randsequence` leads to the call of dedicated `*_body()` tasks. The implementation is left to the end user in a dedicated specialization class.

Users could therefore implement actions to perform the move from one state to another, or to implement actions to be taken when a state is reached.

We rely on the class inheritance and the polymorphism features of the language to extend the provided template. Compared to other approaches where users have to modify and fill placeholders in template files, this simplifies the update of the graph as the user code would remain unchanged. Users would only have to implement missing *_body() tasks to suit their needs.

In case of graph updates, some nodes may disappear and the implemented *_body() task in the user class would not be called. We recommend users to call the related super.*_body() task from their implementation so that if tasks are updated and removed, a syntax error would be reported.

This approach improves the maintenance of the graph as only the graph part is regenerated, while the user class remains unchanged with no risk of being overridden.

For readability, the graph part with the *_body() tasks are separated from the base class which implements all the utility functions and instantiates the core variables.

Additionally, a *checker* class is provided, allowing users to check transitions from their scoreboards or monitors.

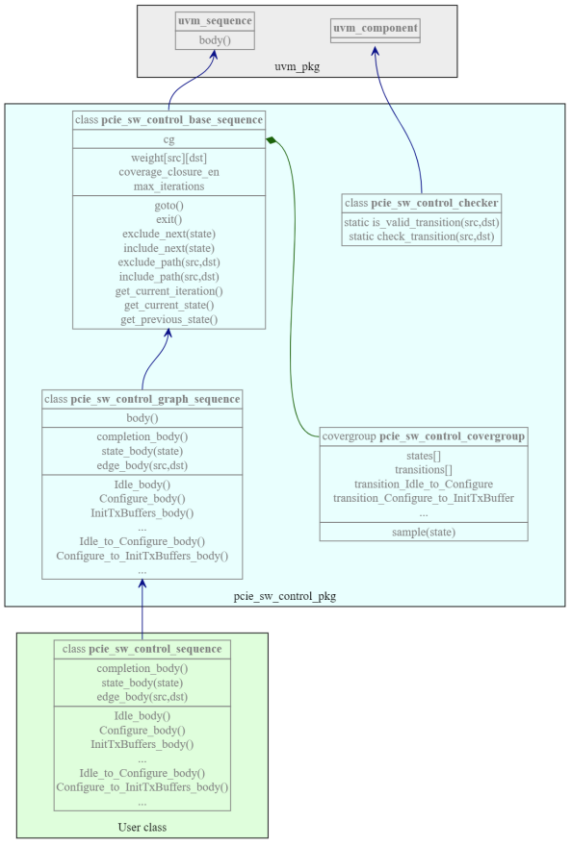

Figure 6 Class Diagram

B. *Edge and State Separation*

The randsequence graph has no intrinsic knowledge of the design's state, so it's up to the final user to implement the required actions for the right *_body() task. Two different understandings are possible:

- State productions are used for actions to make the design move to this state, so the graph represents the target state.
- The actual actions to move from one state to another should be taken on the edge of the graph.

In the first case, actions should be taken to reach a specific state. This could require users to know what the previous state was, so a function get_previous_state() is provided.

In the latter understanding, users would need to have access to placeholders for both **edges** and **states**. Actions on **edges** will be taken once randsequence has been evaluated to move from one state to another, while the state placeholder could then be used to check some design behavior, poll registers, synchronize or other actions related to the state that has been reached. Additionally, in some cases, such as when polling registers or waiting for events, it is required to control to which state we should go next. This must be done from the state and before the action has been decided.

A clear separation between the states and the edges is therefore required to provide the right types of placeholders. The following randsequence template (Figure 7 and Figure 8) provides placeholders for both state and edge action blocks for any type of actions.

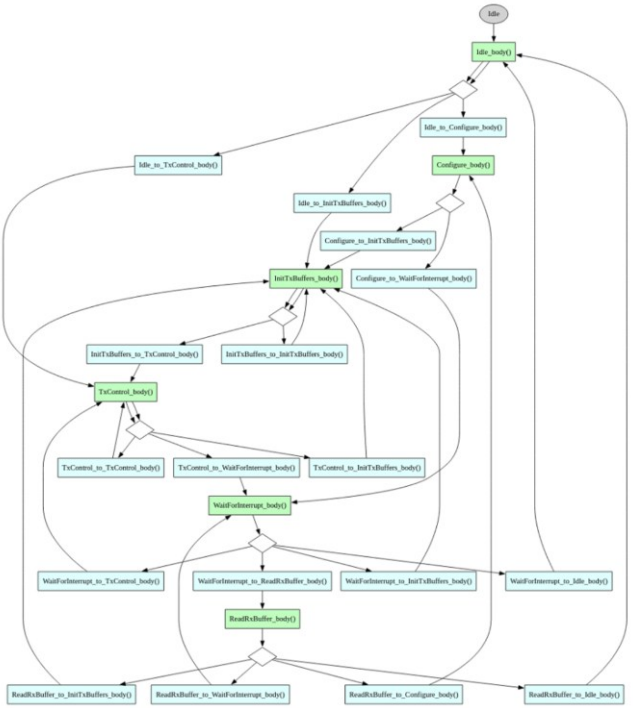

Figure 7 - State/Edge diagram with *_body() placeholders

It also clearly separates the actions from the selection, for better readability and maintenance.

```
randsequence(Idle);
    Idle        : Idle_act           Idle_sel;
    Configure   : Configure_act      Configure_sel;
    InitTxBuffer: InitTxBuffer_act   InitTxBuffer_sel;
    //...
    Idle_act     : { Idle_body(); };
    Configure_act: { Configure_body(); };
    //...
    Idle_sel     : Idle_to_Configure         | Idle_to_InitTxBuffer           | Idle_to_TxControl;
    Configure_sel: Configure_to_InitTxBuffer | Configure_to_WaitForInterrupt;
    //...
    Idle_to_Configure:         { Idle_to_Configure_body(); }          Configure;
    Configure_to_InitTxBuffer: { Configure_to_InitTxBuffer_body(); }  InitTxBuffer;
    //...
endsequence
```

Figure 8 - State/Edge randsequence template with *_body() placeholders

Let's follow this pattern: the randsequence starts in the Idle node. This Idle node is the production of an action Idle_act and a selection Idle_sel. So, the action Idle_act is executed first, leading to the call of the task Idle_body(). As this is a leaf production, we get back to the Idle node which terminates with the production of Idle_sel. Idle_sel will select one of the Idle_to_Configure, Idel_to_InitTxBuffer or Idle_to_TxControl production statementLet's assume that Idle_to_Configure is selected. As a sequence production, it first calls the Idle_to_Configure_body() task, before moving to the Configure node.

We have therefore executed, in order, the following tasks: Idle_body(), Idle_to_Configure_body() and Configure_body().

C. *Controlled Weighted Choices and Exit Conditions*

By default, all choices are equivalent and have the same weight. This can be quite limiting for random testing, could dilute the probability of reaching far end branches, and may lead to infinite production diagrams.

```
randsequence(Idle);
  //...
  Idle_sel          : _END_:=weight_end | Idle_to_Configure         :=weight[Idle][Configure];
  Configure_sel     : _END_:=weight_end | Configure_to_InitTxBuffers:=weight[Configure][InitTxBuffers] //...;
  InitTxBuffers_sel : _END_:=weight_end | InitTxBuffers_to_TxControl:=weight[InitTxBuffers][TxControl];
  _END_: { completion_body(); }
Endsequence
```

The use of variable weights, defined in a constrainable array at the base class level allows the user to be more precise about the graph's behavior.

```
constraint my_user_constraint {
  weight[Idle][InitTxBuffer] == 10;
  weight[Idle][Configure]    == 90;
}
```

Using these weights, allows to implement utility features, required for a complete controllable graph-based verification:

- Exit condition, typically when 100% of the graph coverage is reached, or when a maximum number of iterations occurred.
- A goto() function, that allows users to force the next transition. This is typically required when the FSM behavior is not entirely controlled by the random choices, but also depends on design's behavior, such as interrupts or register status.

D. *Reusable Coverage Model → KPI*

A coverage model is provided using both state coverage and transition coverage.

Since users may decide to restrict some transitions, and still want to perform coverage closure, this coverage should allow disabling specific transitions. However, transition coverpoint bins cannot be deactivated individually. We therefore opted to implement individual coverpoints for each transition.

```
covergroup sw_control_cg with function sample(sw_control_state_t current_state);
  // state coverage
  states : coverpoint current_state {
    bins state      [] = { [current_state.first():current_state.last()] };
    ignore_bins unknown = { UNKNOWN };
  }

  // Global Transition Coverage
  transitions : coverpoint current_state {
    bins transition[] =
       (Idle          => Configure)
      ,(Configure     => InitTxBuffers)
      ,(Configure     => WaitForInterrupt)
      ,(InitTxBuffers => TxControl)
      //...
      ,(TxControl     => TxControl)
      ,(TxControl     => InitTxBuffers);
  }

  // Individual Edge Coverpoints, allow fine grain objective setup
  Idle_to_Configure : coverpoint current_state {
    bins Idle_to_Configure = ( Idle => Configure );
  }
  Configure_to_InitTxBuffers : coverpoint current_state {
    bins Configure_to_InitTxBuffers = ( Configure => InitTxBuffers );
  }
```

This covergroup is provided as an independent covergroup and is instantiated in the main sequence class. The separation from its implementation allows users to reuse this covergroup from their scoreboard or monitors.

E. Checkers

The *checker* class implements a transition check based on the provided graph. Although this implements exactly the transitions that are made possible from the randsequence crawl and has little interest from the sequence itself, this can be easily reused by users to implement checkers in their scoreboards or interfaces, thus making sure that the design behavior is aligned with the test intent.

```
function void pcie_link_training_checker::check_transition(pcie_link_training_state_t src,dst);
    ASSERT_PCIE_LINK_TRAINING_TRANSITION: assert (is_valid_transition(src,dst))
    else `uvm_error("ASSERT_PCIE_LINK_TRAINING_TRANSITION",
                    $sformatf("Invalid transition from %s to %s",src,dst))
endfunction : check_transition
```

V. AUTOMATIC GENERATION OF THE VERIFICATION PATTERN

Knowing a state-machine or having in mind any graph that represents the chain of possible test sequences, it is now easy to implement a tool which generates the randsequence, the class with the virtual empty *_body() tasks, the covergroups and the checkers. A simple CSV with one column with the original state and one column with the destination state is sufficient to fully define the generated code. The script also supports inputs as Graphviz dot files and outputs as csv, dot, uvm and standalone SystemVerilog randsequence.

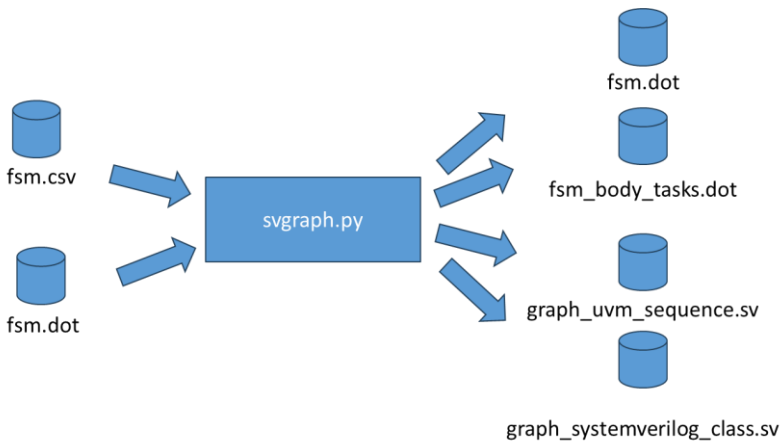

Table 3 - Generation Flow

VI. USE CASE AND RESULTS

An open-source simplified version of the generation script is available on gitlab and an online form using it is available on our company website [4].

We have first used the described randsequence approach on a complex programmable PWM verification, defining the PWM grammar using dedicated UVM sequences with randsequence in their body() and a higher level UVM sequence calling them in a randcase. This proved to gain in project productivity and overall documentation of the tests.
The current version of the script has been used to generate the sequences of a few other designs, including the ARP management of the SMBus protocol and inside our PCIe Verification IP.

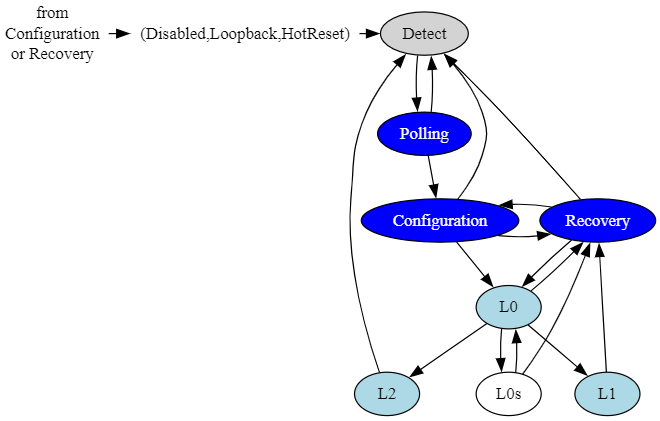

Figure 9 - PCIe Link Training FSM

The PCIe training sequence relies on layered state-machines, where actions should be taken on each state and on each transition, and where each transition choice is either made by the receiver or by the transceiver. Whether we consider the Receiver part or the Transceiver part of the VIP, this leaves a certain freedom for the VIP to make random decisions, while on some other occasions the VIP has to react to the DUT.

A simplified version of the Link Training and Status State-Machine is represented in Figure 9.

This has been quickly implemented as an FSM transition table in a CSV file (Table 4 - State Transitions CSV File).

And within minutes the main UVM sequences being generated with the core body using the randsequence as shown in this code snippet:

```
    task aedv_pcie_link_training_base_sequence::body();
      randsequence(Detect)
        Detect : Detect_act Detect_sel;
        Polling : Polling_act Polling_sel;
        Configuration : Configuration_act Configuration_sel;
        L0 : L0_act L0_sel;
        Recovery : Recovery_act Recovery_sel;
        L1 : L1_act L1_sel;
        L2 : L2_act L2_sel;
        L0s : L0s_act L0s_sel;

        Detect_act : {
          prepare_context();
          fork Detect_pre_body(); state_pre_body(Detect); join
          fork Detect_body(); state_body(Detect); join
          fork Detect_post_body(); state_post_body(Detect); join
        };
//...
```

The actual sequence then implements the VIP actions in the different state and edge body sequences, creating the dedicated physical layer transactions depending on the current states, or on the destination state.

state	next_state
Detect	Polling
Polling	Detect
Polling	Configuration
Configuration	L0
Configuration	Recovery
Configuration	Detect
L0	L1
L0	L0s
L0	L2
L0	Recovery
Recovery	Configuration
Recovery	L0
Recovery	Detect
L1	Recovery
L2	Detect
L0s	L0
L0s	Recovery

Table 4 - State Transitions CSV File

Some of these choices are made by the Tx, while other transitions depend on what is received on Tx. So, the goto() action must be used.

Another aspect of this state machine is that it is defined as a top-level state machine, with lower-level state machines for each node. For example, the Polling node then defines lower-level choices, which can also be made using the graph-based approach.

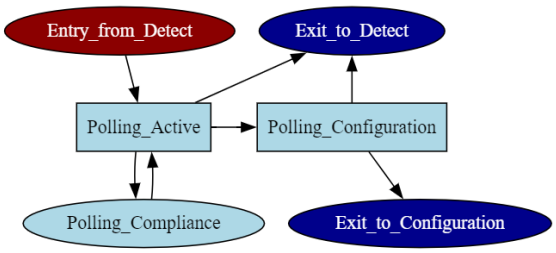

Figure 11 Generated UVM tasks and randsequence

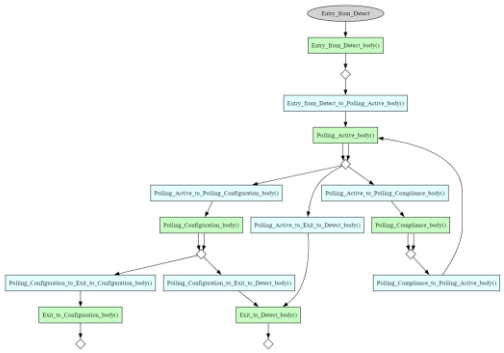

Figure 10 Link Training FSM Polling Sub-States

The upper layer sequence can therefore call this sub-sequence in the corresponding polling_body() sequence and then use the goto() function to jump to whatever state has been decided in the sub-sequence.

```
class aedv_pcie_link_training_sequence extends aedv_pcie_link_training_base_sequence
  task polling_body();
    pcie_training_polling_seq seq;
    seq.start();
    if ( seq.state == exit_to_detect )
      goto(Detect);
    else
      goto(Configuration);
  endtask
endclass : aedv_pcie_link_training_sequence
```

Overall, this approach reduces to minutes the development of complex state machine sequences, and to hours or days the implementation of the corresponding actions when days and weeks were necessary prior this generation script.

VII. CONCLUSION

In this paper, we presented a *pattern* to use the SystemVerilog randsequence statement to accelerate the development of complex state-machine verification. This pattern encapsulates the full FSM graph as a single UVM sequence and provides facilities for automatic coverage closure, user actions and checking. This facilitates the creation of fully random test scenarios, removing the needs to implement complex code, and easing the ramp-up of junior engineers on such verification. In future evolution of this development, investigations will be made to automate further FSM layering as in PCIe Link Layer and automatic shortest path crawling of the graph to accelerate coverage closure on big state-machines.

REFERENCES

[1] IEEE Std 1800™ 2023, IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language

[2] Unified Approach to Verify Complex FSM (https://verificationacademy.com/verification-horizons/november-2020-volume-16-issue-3/unified-approach-to-verify-fsm/)

[3] Modelling Finite-State Machines in the Verification Environment using Software Design Patterns, Darko M. Tomušilović, Mihajlo Z. Minović, DVCon 2017

[4] https://gitlab.com/aedvices_pub/gbug and https://aedvices.com/en/gbug/

[5] **Testing Resource Isolation for System-on-Chip Architectures**, Ledent et al., *Proceedings Sixth Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2024, Luxembourg City, Luxembourg, 6th April 2024* , EPTCS Vol. 399 p. 129-168 (https://doi.org/10.4204/EPTCS.399.7)

[6] Improving PSS Test Generation Using Model Checking and Conformance Testing, Ledent et al., *Forum on Specification & Design Languages, FDL 2024, Stockholm, Sweden, September 4-6, 2024* (https://doi.org/10.1109/FDL63219.2024.10673842)

[7] Portable Test and Stimulus Standard v.3.0, Accellera Systems Initiatives, 2024, (https://accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v3.0.pdf)