Journal of Electronic Testing
https://doi.org/10.1007/s10836-020-05866-5

Automated Bug Resistant Test Intent with Register Header Database for Optimized Verification

Gaurav Sharma[1] · **Lava Bhargava**[1] · **Vinod Kumar**[2]

Received: 25 September 2019 / Accepted: 19 February 2020

Abstract

The biggest challenge in the verification industry is to create a sufficient number of valid test cases to acquire the desired coverage closure. The design complexity is relentlessly increasing with the number of gates, IPs, embedded processors, software content, and many more. It diverts the research once again to a point, where verification needs a productivity boost to cope with increasing design complexity. The extensive SoC design verification processes are using different execution platforms like simulation, emulation and FPGA prototyping. Each of these platforms requires different ways of specifying tests. The productivity and time demand a single test intent to reuse it across all verification execution platforms. Universal verification methodology (UVM) provides a verification efficiency jump from directed tests to constraint random tests. Synthesis allows the design productivity to jump from gate level to RTL level. Similarly, the SoC verification needs a higher-level of abstraction for automation enactment across all execution platforms. The proposed work derives a test intent on a UVM register abstraction testbench to generate automated test cases. It also assists in the improved functional coverage metric through bug resistant algorithm. The testbench also saves the manual effort of writing test cases. The work improves the simulation time, CPU time, and functional coverage closure in lesser number of transactions as compared to state of the art test benches.

Keywords Universal verification methodology · Test intent · Test case automation · Bug resistant

1 Introduction

Verification is the longest and most critical phase in the design development cycle. Almost 50-70% effort in total design cycle time goes to verification [1]. The verification process supports different transactions, remove bugs and errors introduced during the process, and cross-platform compatibility. SoC verification process needs an architecture boost to cope up with the present complex SoCs. The increasing amount of content on a single chip challenges its verification time and capability. Creating the proper number of test cases for SoC verification is always challenging and time-consuming. Manual intervention in test case writing opens up the provision for error and bugs in tree-based verification models [2, 3]. Manually writing test cases is a very challenging task. In our previous work, we have implemented an automated register database creation framework to adapt testbench for faster simulation [4]. The design register database random stimulus verification inside the testbench environment hastens the simulation as compared to conventional SystemVerilog and UVM testbenches. The RAL model by UVM lacks in automation and test case generation for complex SoC designs. The RAL model is a general verification technique lacking coverage, bug removal mechanism, header file creation and test case generation. With time the old UVM RAL model has been improved, but still, it gets short of some feature due to increasing SoC complexity. The RAL model needs a regular up-gradation to cope up with the originating demand of complex SoC modules. The proposed work provides significant coverage, bug removal mechanism, header file creation and test case generation, which helps the conventional UVM-RAL model to handle intricate designs. The automation scripts

Responsible Editor: N. Nicolici

✉ Gaurav Sharma
2015rec9014@mnit.ac.in

Lava Bhargava
lavab@mnit.ac.in

Vinod Kumar
vinod.kumar_1@nxp.com

[1] MNIT Jaipur, India

[2] NXP Semiconductors, Noida, India

Published online: 14 April 2020

may generate the test cases, but it also opens the provision for error intervention. C test relies on UVM to drive external stimulus, so scoreboards and checkers are implemented in UVM. Mostly C tests are directed tests through UVM. C directed test cases lower the accuracy and miss the corner cases. UVM has certain performance and time-related issues for complex system-level tests [5]. These limitations generate a need for a general test intent which can be used across multiple platforms. The work proposes a test intent automated architecture to handle cross-platform communication. The architecture also adjoins the proposed bug resistant algorithm for generating numerous bug-free test cases. The approach automates the reusable test intent for scenario level tests. The model supports the use of C and UVM both to extract their benefits. Design specifications are in domain specific language (DSL). DSL compiler creates a scenario model database. C++ library is added to generate the same scenario model which has to be generated through DSL compiler [6]. A design must perform and pass the model comparison to create scenario model database. Test automation block generates C tests through the main model database, followed by a bug resistant algorithm. UVM testbench generates random cyclic sequences to run test cases on design under verification (DUV) [7]. These cyclic sequences are non-repeatable to improve verification time and code coverage. The architecture uses two test benches. One is UVM testbench, and the other is SoC testbench. UVM testbench peruses module by module verification whereas SoC testbench verifies complete SoC using appropriate test cases. The generated test cases are the possible combinations of single/multiple write or read on appropriate module/registers/wire on random and specific addresses. The main perspective is to hit all coverage bins in the minimum number of iterations. If a logic bin is hit once, it shouldn't be hit again, because its verification is over. This approach fastens the verification environment, makes the flow capable to hit all logic bins and needs much lesser iterations to complete the run. The bug resistant test case automation improves the coverage with all bin hits in the minimum number of transactions as compared to the conventional UVM constraint random architecture. The bug resistant process significantly reduces the simulation time and CPU time for SoC verification.

The proposed architecture includes cross-platform automated test intent, constraint random coverage directive test intent, and bug resistant features. The cross-platform test intent allows the testbench to create test cases for a different environment. The constraint random coverage directive test intent improves the functional coverage and significantly contributes to decreasing the simulation time by shrinking the number of program iterations. The bug resistant algorithm works on scenario level modeling; it is used to overcome the performance and corner case issues of C tests [8]. The algorithm is designed to find and clear hidden bugs including corner cases associativity.

The number of registers is also increasing with the SoC complexity adhere to the memory and address signals. Every register has its contention. Some registers consist of mask bits, and some registers are unmasked. The verification process of these register contentions becomes very critical along with the increasing number of registers. The conventional UVM constraint random verification environment consumes high CPU/ process time [9, 10]. The UVM constraint random architecture also has functional coverage productivity issues due to randomized transactions [10]. The conventional component wrapper language model and constraint random hits the bins multiple times with more number of required transaction [6, 10, 11]. The lack of covergroup traces leads to disabling the attempt for some coverpoints in matric driven verification methods [12]. This process tends to decrease the cross-functional coverage. Due to the lack of scenario traces in conventional phenomenon, some bins hit multiple time, and some bins remain unattended. It predicts the repeatability of scenario traces consuming extra simulation cycles [13]. The simulation cycles are a nonrenewable resource. The motive of work is to develop a non-repetitive cyclic transaction scenario traces to improve functional coverage and minimize simulation cycles.

2 Test Intent Architecture Implementation

There are certain aspects of generating test cases using test intent. One begins with the specification of a scheduling graph, abstract behaviors and actions to verify the design, keeping the data flow and constraints under proper rational logic. The first part of test implementation is to supply target specific applications for each abstract behavior, including optional code generation for the ultimate test implementations. The second step is diverted through the script to automate different scenarios by stitching together the pieces of implementation code associated with the elements of test intent model [3]. The way assures that the resulting code matches the scheduling semantics of the abstract model. Fig. 1 shows the proposed test intent testbench architecture. The test intent supports domain specific language (DSL) and C++ library to describe the abstract scenario model. The DSL model is compiled by DSL compiler to produce a data structure consisting of all the elements and constraints in the model. It also includes the set of packages in the other available elements. DSL compiler is a python class declaration script. The test intent is compatible with the specialize C++ library [14]. The C++ library is semantically prevalent to DSL, to produce the same scenario model structure. The model extends through

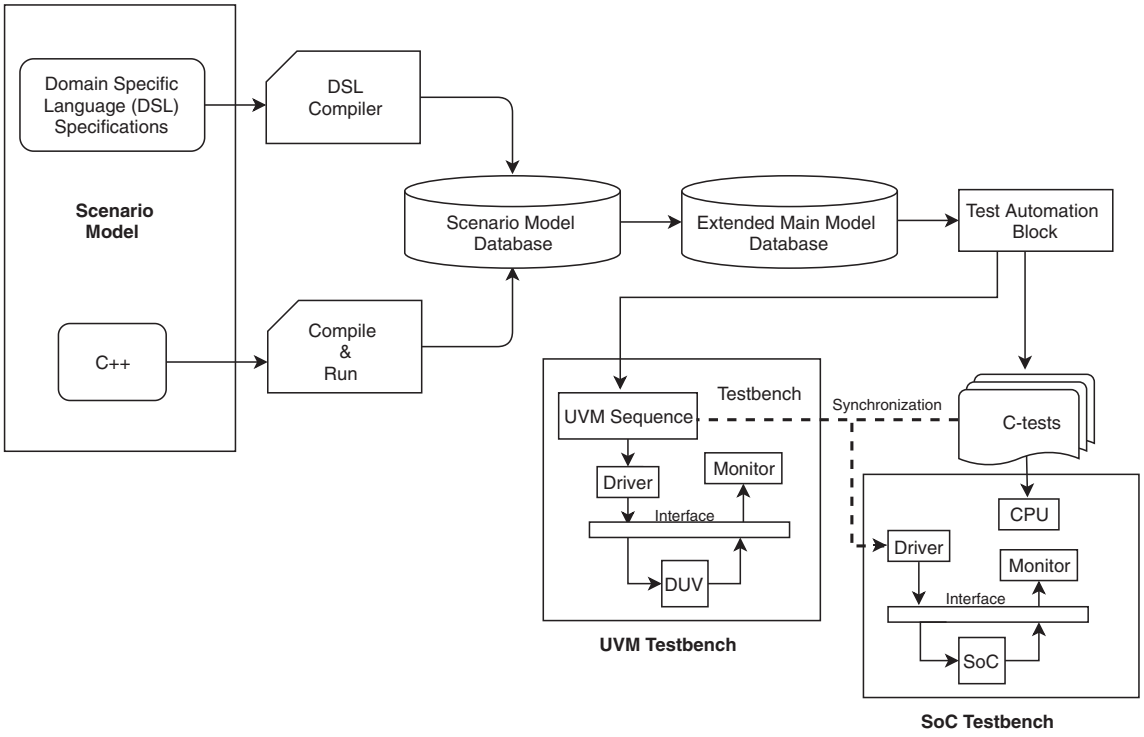


Fig. 1 Test intent architecture flow for execution of automated test cases

solving all the scheduling and other constraints to create the main model. The main model generates all the possible scenarios defined by the original abstract model. The test automation block applies the specific implementation of the scenarios to produce a UVM sequence calls for a fully UVM environment. The block also generates a C test code for an environment to run on the SoC testbench, as shown in Fig. 1. The C code requires a synchronization code to coordinate with UVM sequence to accomplish individual Transaction-level modeling (TLM) tasks under the application program interface (API) [8, 11].

The high-level abstract DSL/C++ scenario model consists of the actions and activities, as shown in Fig. 1. The scheduling of multiple scenarios opens the way to explore multiple test scenarios. The standard semantic language model is created to attain a static scheduling structure in the scenario model database. A simplification platform solves the constraints and schedules in the final static scheduling structure to create a C test intent implementation in the extended main model database. The model is mapped to a C skeleton to create an executable solution for that particular C code scenario in the test automation block. Activity graph is mapped to C code to call the actions in order of their schedule as virtual sequence for UVM and threads for C. The call of actions is specific in terms of actions of correct type, actions supporting downstream flow, set of actions producing correct stream and resource conflicts for scheduling. The final body scenario is mapped to the C-header file, function declaration and main test-tasks skeleton, as shown in Fig. 2.

The specification phase starts with the declaration of covergroups and sequence items. The task execution phase includes a UVM platform that consists of sequencer, driver, monitor, interface, and DUV, as shown in Fig. 3. The covergroup module measures coverpoints and cross-coverage between coverpoints inputs. All scenario traces of a particular coverpoint helps to trace some uncovered scenarios, bins and specific test cases to hit that uncovered scenario. The coverpoints and specifications traverse the test automation model. User specifications and sequence item determine the automated sequence class in the specification phase. The test automation block includes the coverage strategy to determine the exact trace of scenario level graph instances to hit coverage goals. The result concludes with a huge tree-based scenario level graph [2]. The scenario graph covers all the possible coverpoint traces of SoC. The specification phase generates the automated sequence with underlying data structures to establish communication with UVM testbench. The transaction propagates through running the sequence on UVM sequencer. Test Intent For SoC level verification aims to set the configurations modes and covergroups to achieve operational goals. The test automation block in Fig. 1 reads the sequence items and covergroups to create test cases internally that targets the same coverage and constrains as specification phase. The auto generated stimulus directly runs on the UVM testbench. The direct sequence run attains the higher-level abstraction sequence rather than just randomizing the individual transactions like UVM constraint random [15]. The

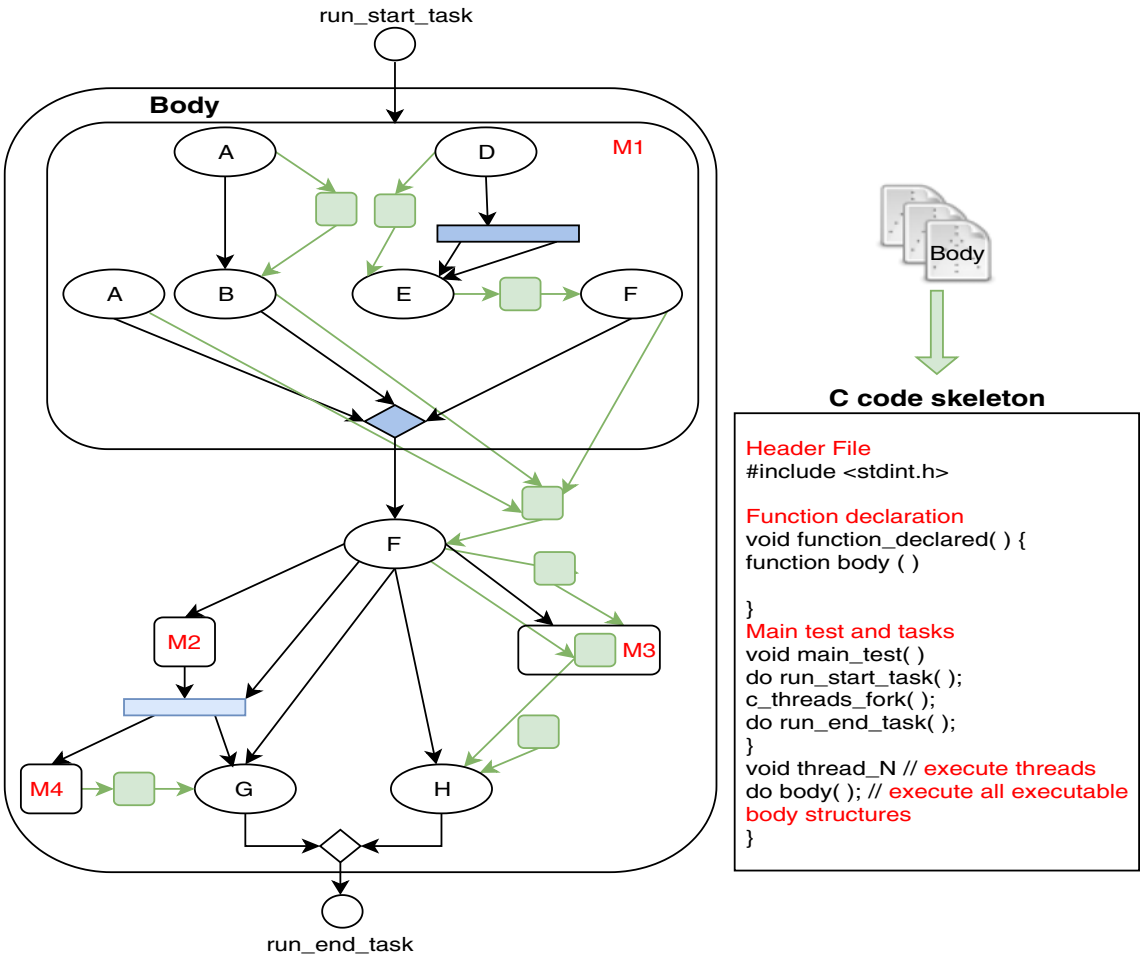


Fig. 2 Abstraction activity mapping to C code skeleton

higher abstraction leads test intent architecture to compose higher-level scenarios to accumulate multiple transactions at different interfaces. All transactions automatically coordinate with the testbench according to the scenario trace.

A scenario model supports two important elements actions and activities. Actions define the behavioral execution and, activities define the ordering in the scheduling of the graph in which actions execute. The abstract scenario model converts the abstract model actions to uvm_sequence and

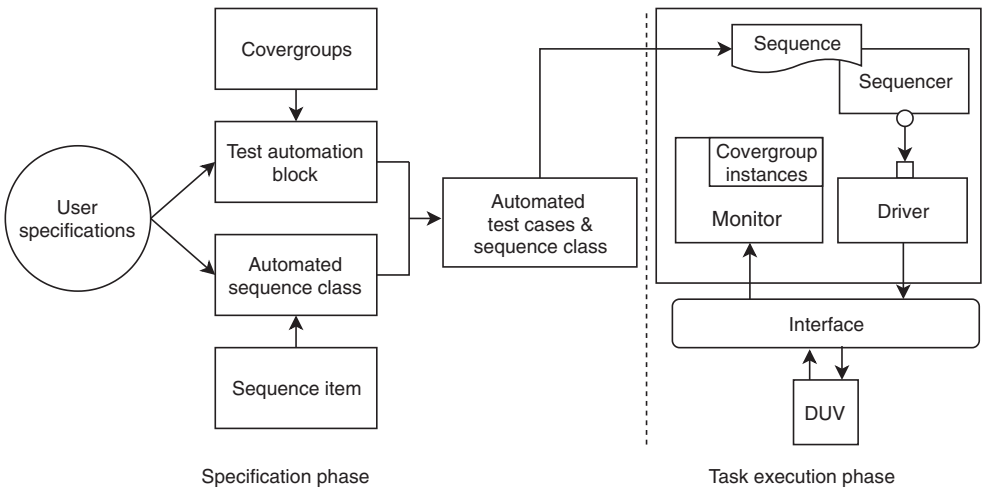


Fig. 3 Specification and test execution phase for propagation of test cases and sequence class

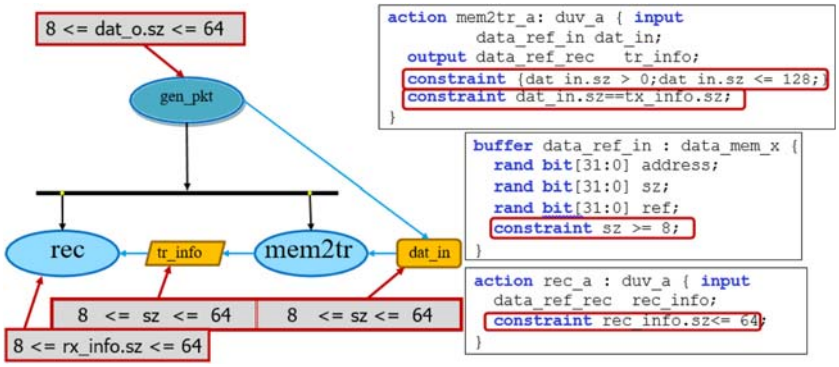


Fig. 4 Constraint insertion on data set parameters

activities to virtual_sequence. The generated sequences and virtual sequences use the UVM top/test/environment/agent testbench architecture, as shown in Fig. 1. The sequences are executed on agents. The block combination of sequencer, driver, and monitor is called agent [4]. The test calls virtual sequences to map the test intent compound actions to get the final scheduling [16]. It is essential for target implementation to attain the data structure compatibility to test intent. The packet generation module generates a data buffer data_ref_in as depicted in Fig. 4. This buffer works as an input to the data check module. Fig. 4 highlights the data constraints and their respective module application in blocks.

Data structure selection is an important aspect in test intent generation. The packet generation module generates data_main_buffer and the data check module uses the same data packet as an input as shown in Fig. 5. data_gen_tx_rx is transmitted and received data. The work is to generate and check the data. Allocation of data needs two randomize fields in a struct address (addr) and size (sz). These are the basic information needed to acquire data somewhere from memory. data_buffer_top construct is extended through object-oriented inheritance to add in gen field, which is also random as shown in code below.

```
struct data_buffer_top {
    rand bit [31:0] addr;
    rand bit [31:0] sz;
}
buffer data_main_buffer:
data_buffer_top{
    rand bit [31:0] gen;
}
stream data_gen_tx_rx {
    rand bit [31:0] addr;
    rand bit [31:0] gen; }
```

Similarly, transmitted (tx) and received data (rx) also share a random data stream. Since the data stream can be stored anywhere, the address field is not required here.

The test intent defines the functional coverage points to illustrate the critical states in verification [17]. The constraint random tests hit the critical states in random ways. The problem associated with constraint random test is that it may repeat some states or miss some coverage points [6, 10]. The identification of a test is the track of an isolated test path. Thus, a python script works for test path isolation to provides clarity in random test functional coverage. There is a probability that a bug might hide outside the set of states in the activities. The activities are defined in an explicit scheduling graph. The test intent integrates the functional coverage in the test generation. Thus, it uses the functional coverage to depict the critical verification states [15]. The test intent specifications are declarable; it analyses the possible scenarios in the graph and generates the liable tests for hit on coverage points. This test intent coverage scripting provides the maximum coverage in a very lesser number of tests. It amicably generates all tests to hit the coverage points in all possible combinations. The activity graph is declarable, So it generates all possible paths from the full graph to guarantee the hit to all coverage points with additional states to uncover hidden bugs, as shown in Fig. 6.

The ability to automate the multi-layer scheduling tree is an important step in the direction of building intelligent verification [2]. The goal is to make a fast and bug resistant testbench. The multi-layer tasks are very complex, and it can compute many things with proper assistance. The script begins with the threshold setting for each test intent testbench module for code and functional coverage. These threshold values are stored in terms of weights to execute the flawless code flow. I_i, H_i and O_i represents the testbench module inputs, hidden bugs and module outputs as depicted in Fig. 7. The threshold connections between inputs and hidden bugs are denoted by $W0_{i,j}$, while

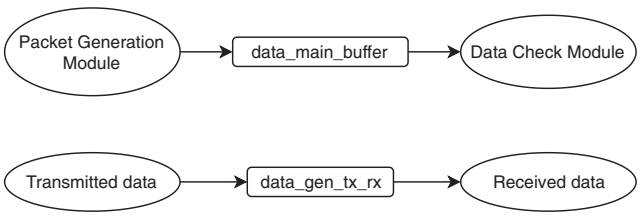


Fig. 5 Data types- data generation and data check module diagram

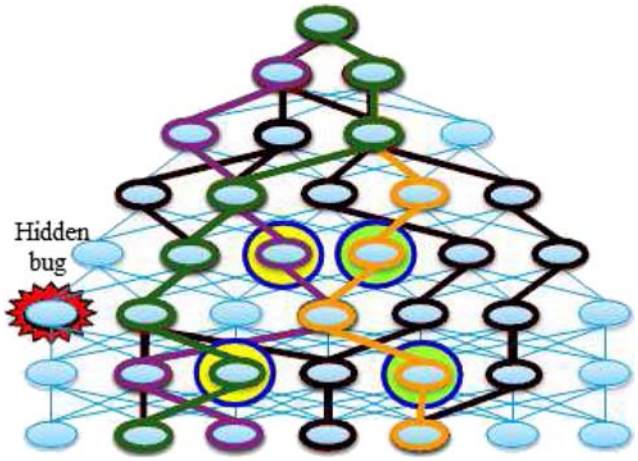


Fig. 6 Scenario level tree graph consisting multiple scenarios to find hidden bugs

threshold connection between hidden bugs and module outputs is $W1_{i,j}$. The considered scheduling scenario model has three layers; it may be more. The activation flow is from inputs to hidden bugs to outputs in the forward direction. The hidden bugs may activate through vertical or diagonal propagation of test schedule.

Algorithm:

1. Let X be the number of input and Z be the number of outputs. Now identify Y, the number of hidden bugs. The inputs and hidden bugs has used one extra unit of threshold indexed as (0....X) and (0....Y). The theorem denote I_j, H_j and O_j as activation levels of inputs, hidden bugs and outputs.
2. Initialize the threshold in network. If the coverage threshold set is between .8 to 1 (80% to 100%).

$$W0_{i,j} = random(.8, 1) \quad \forall i = 0, ..., \mathrm{X}, \mathrm{j} = 1, ..., \mathrm{Y}$$
$$W1_{i,j} = random(.8, 1) \quad \forall i = 0, ..., \mathrm{Y}, \mathrm{j} = 1, ..., \mathrm{Z}$$

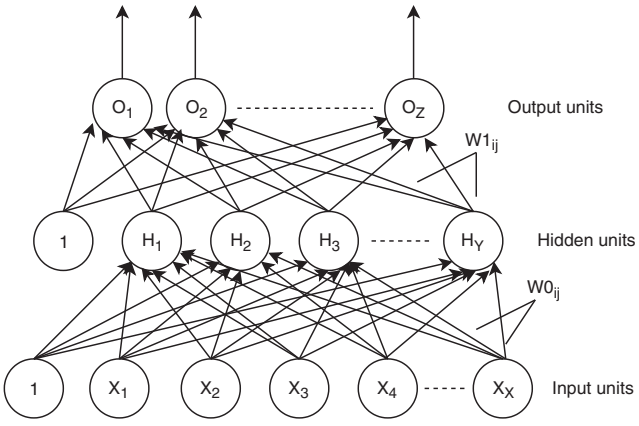


Fig. 7 Bug finding algorithm hierarchy to compute acceptable functional coverage

3. Initializing the activity of threshold units. The threshold values remain to fix during automation.

$$I_0 = 1.0; H_0 = 1.0$$

4. Choose an input-output pair. Assign activity levels to input vectors.
5. Propagate the activity from the input layer to the hidden bug layer using activity function as shown in Eq. 1.

$$H_j = \frac{1}{1 + e^{-\sum_{i=0}^{X} W0_{i,j} I_i}} \quad \forall \ j = 1, ..., Y \tag{1}$$

6. Propagate the activities in the hidden bugs layer to output layer as shown in Eq. 2.

$$O_j = \frac{1}{1 + e^{-\sum_{i=0}^{Y} W1_{i,j} H_i}} \quad \forall \ j = 1, ..., Z \tag{2}$$

7. Compute the errors in the output layer denotes as $\delta 1_j$. bugs are based on the testbench actual output (O_j) and the target output (Y_j) as shown in Eq. 3.

$$\delta 1_j = O_j(1 - O_j)(Y_j - O_j) \quad \forall \ j = 1, ...Z \tag{3}$$

8. Commute the errors in hidden bugs layer as $\delta 0_j$ as depicted in Eq. 4.

$$\delta 0_j = H_j(1 - H_j) \sum_{i=0}^{Z} \delta 1_i \cdot W1_{j,i} \quad \forall \ j = 1, ...Y \tag{4}$$

9. The automation function α keeps balancing the threshold until it achieve the perfect threshold for test intent schedule and coverage as shown in Eq. 5.

$$\Delta W1_{i,j} = \alpha \cdot \delta 1_j \cdot H_i \quad \forall \ i = 0, ..., Y, \ j = 1, ..., Z \tag{5}$$

10. Again adjust the threshold between input layer and hidden layer as shown in Eq. 6.

$$\Delta W0_{i,j} = \alpha \cdot \delta 0_j \cdot I_i \quad \forall \ i = 0, ..., X, \ j = 1, ..., Y \tag{6}$$

11. Jump to step 4 and repeat steps from 4 to 10 until all possible combinations of test intent scenarios are completed.

3 Definition of Scenario Level Test Cases

This section introduces the proposed formal scenario level test case generation. The activity diagram for the scenario level formal verification is represented as follows:

3.1 Activity Diagram

Equation 7 depicts the relationship of activity diagram with nodes and edges.

$$ACT\_DIAG = (Node; Edge) \tag{7}$$

The node is the collection of all nodes as shown in definition as follows:
Node = (StartNode; FlowTerminationNode; ActivityTerminationNode; ActionNode; ActivityNode; Fork-JoinNode; DecisionNode; MergeNode; ReceiverNode; SenderNode)

The significance is to start from StartNode to any node, it is the beginning node of activity diagram. The path will terminate from any other node to ActivityTerminationNode, the exact end of activity diagram.

The edges in the activity diagram are classified into two categories: token passing edges and data flow edges. Token passing edges shows the process of token passing in the activity diagram. Data flow edges show the data flow in between the different activities in the activity diagram. Equation 8 determines the relationship between the edges and nodes.

$$Edge = (x, y)|x, y \in Node \tag{8}$$

3.2 Test Cases

The test case based on finite state machines, activity diagram and transaction-level modeling consists of test pattern activity scenarios and input data sets. The test cases representation for activity diagrams is depicted in Eq. 9

$$Test\_Case(ACT\_DIAG) = (Test\_scenario, Data) \tag{9}$$

The test scenario activity graph consists of numerous actions and edges. The path of execution is from StartNode to ActivityTerminationNode. The activity diagram defines the test path in form of nodes and edges is shown in Eqs. 10,11,12.

$$path = (a_1', a_2', a_3' ....a_n') \tag{10}$$

$$a_i' = (E_n, a_n), (i = 2, ..., n) \tag{11}$$

$$E_n = a_{i-1} \rightarrow a_i, (i = 2, ..., n) \tag{12}$$

Here a_i is nodes and E_n is edges in the above equations. The test path is a tree graph starts from StartNode a_1 and terminates at ActivityTerminationNode a_n.

3.2.1 Coverage Matrices

The proper set of coverage matrices is necessary for measuring the performance and efficiency of a verification architecture [15, 18]. The coverage criterion of a verification model includes functional coverage, code coverage, cross-functional coverage, statement coverage, branch coverage, conditional coverage and many. The verification architecture needs the following matrices to adjudge the activity diagram.

- **Functional coverage matrix:** The functional coverage matrix elaborate the functional and cross-functional coverage correctness to resolve the module dependencies [10, 19]. It measures the number of transactions to complete the verification, by summing the number of hits on bin 0 and bin 1.
- **Action coverage matrix:** This matrix is responsible for execution of all actions in every test case program at least once during the simulation.
- **Edge coverage matrix:** This coverage matrix assures the execution of all test path edges at least once during the simulation of test cases.
- **Path coverage matrix:** The path coverage matrix maintain the proper execution of all test paths from StartNode node to ActivityTerminationNode.
- **Branch coverage matrix:** This coverage matrix assures the run of all decision branches to execute the instructions in the form of decisions. Some actions may change the branch after a true or false condition.

The above coverage matrices designs the coverage attributes after the simulation to predict that the legal coverage is accomplished or not.

4 Automated Test Case Scenario Generation

Automatic test case scenario generation architecture analyses structured activities, loop activities and concurrent activities. Multiple start node transformation problems adapt algorithm 1 for solution.

4.1 Structured Activities

Structured activities are basic activity structure in the simple scenario shape. Algorithm 2 finds all basic test path scenarios between start and termination node using graph theory. Algorithm 2 describes the automated test path scenarios for activity diagrams. Depth first search (DFS) algorithm transforms structured mode to non-cyclic directed graph. All the test paths are generated from start to termination node in the stack form of storage. The flow utilizes the DFS algorithm without considering the loop and concurrent activities.

Algorithm 1 Solution to multiple start nodes.

Input : Compressed activity diagram
Output: Transformed activity diagram
Data *N_0 zero-degree set of nodes, n_0 number of nodes with zero-degree, set to 0 in start*
Algorithm:
begin
  for *each node* $node_i \in$ *activity diagram* **do**
    if $node_i.branches_i n = NULL$ **then**
      $N_0 = N_0 \cup node_i$;
      $n_0 = n_0 + 1$;
    end
  end
  if $n_0 == 1$ **then**
    return activity diagram;
  end
  else
    new fork node (F);
    for *every* $node_i \in N_0$ **do**
      new create edge from fork node (F) to $node_x$;
      new the start node;
      new the edge initiates from start node to fork node (F);
      return activity diagram
    end
  end
end

Algorithm 2 Scenario (test path) generation for design modules.

Input : Compressed activity diagram
Output: n number of scenarios
Data *Nodes, paths and scenarios*
Algorithm:
begin
  Path.new(start);
  make|$path.start().alledges$|; making replica of all paths
  for *every node in activity graph* **do**
    if $path.start().node_i$ **then**
      get a PathReplica;
      $PathReplica.new(start)$;
    end
    if $node_i = ActivityTerminationNode$ **then**
      $TotalScenario = ExistingScenarios \cup node$;
      delete $PathReplica$;
    end
  end
  if $|PathReplica| \neq 0$ **then**
    Jump to making replica of all paths separately
    $make|path.start().alledges|$; making replica of all paths
  end
  else
    return Scenarios
  end
end

4.2 Loop Activities

The activity diagram in loop activity is the number of cycle repetitions until the conditional acceptance. The condition judgment cycle performs the loop multiple times. Activity diagram is divided into do-while loop and while-do loop. do-while loop runs only one time and while-do loop run n number of times. Loop activity transformation uses a module test path generation algorithm for creating different scenarios. The generated loop scenario replaces loop activity box in the loop activity taking only finite loop iterations in concern. An expanded algorithm is proposed to deal with basic loop, nested loop and unorganized loop. For N iteration basic loop, following steps process the basic loop activity test cases.

- Jump the complete loop.
- one iteration of loop.
- Two iteration of loop.
- k iterations of loop.
- N-1, N, N+1 iterations of loop.

It is inappropriate to use basic loop steps in nested loop. It produces very large number of test cases with increasing in layers of nested loops, impractical to execute.

- Analyze the iterations in innermost loop and set the number of cycles to it.
- Test the innermost loop keeping the outer loop to minimum cycle iterations.
- Continue to test all the loops from innermost to outer loops keeping the outer loops iteration to minimum.

The unorganized loops are the combination of basic loops and nested loops. We try to structure the unorganized loop activities using basic and nested loop techniques.

4.3 Concurrent Activity

The concurrent activity introduces the fork and join node concept. Fork node illustrates the simultaneous generation of parallel data streams. Join node depicts the possibility of joining these parallel streams at ActivityTerminationNode.

The Fork representation of this activity is logical ”AND” gate, as it waits until for the complete parallel data steams before transferring into the next action. However, the join node works as logical “OR” gate as it waits to merge all synchronized stream flow at ActivityTerminationNode to proceed for next action execution.

The scenario transformation is join dependent. The number of parallel synchronizing streams are divided into subsets. The join merge node is made as separate join for synchronized parallel streams. Some streams may need no parallel join merge. Basic concurrent loop uses test path generation algorithm directly to produce test cases. While solving the concurrent merge loop, semi join concurrent loop and non join concurrent loop, simple test path scenarios are produced by solving the concerning loop activities. These simple test path scenario replaces activity loop box in activity diagram. Concurrent merge loop follows the ”OR” logic operation.

The automatic test case scenario generation algorithm for concurrent merge loop is depicted below:

- Execute an optimization algorithm for basic concurrent loops to create a test path.
- Copy all the concurrent flow stream nodes to one parallel data stream of test path scenario and add merge node in the test path scenario.
- Reject the activity nodes in other concurrent data streams.
- Accomplish a single test path for concurrent merge loop module.

The automatic test case scenario generation algorithm for semi join concurrent merge loop is depicted below:

- Execute an optimization algorithm for basic concurrent loops to create a test path.
- Copy all the concurrent flow stream nodes to one parallel synchronized data stream, which requires the synchronization with the generating test path to direct the flow towards the join node. A join node is added in the test path.
- If some nodes don’t need synchronization, and are not copied in the test path scenario, the algorithm works until the coverage of all nodes in test path scenario.
- Accomplish a single test path for semi join concurrent loop module.

The automatic test case scenario generation algorithm for non join concurrent merge loop is depicted below:

- Execute an optimization algorithm for basic concurrent loops to create a test path.
- Copy all the concurrent flow stream to one parallel synchronized data stream of test path scenario. If the copy consists of ActivityTerminationNode, it indicates the completion of the activity. It rejects the other nodes in concurrent streams.
- Copy of ActivityTerminationNode decides the completion of one concurrent activity. The algorithm runs in a loop until it completes all left concurrent activities.
- Accomplish a single test path for join concurrent loop module.

4.4 Problem: Multiple Start Nodes for Action Execution

The work proposed the algorithm as a solution space for multiple start node to act in the activity diagram. The proposed algorithm works in a way to create n concurrent paths out of n multiple start nodes. These n concurrent paths are formalized into n loop activities to transform it to the scenario level activity diagram. Algorithm 1 depicts the transformation solution to multiple start nodes in the activity diagram. The algorithm solves the multiple start structure to the similar structural flow of one start node. Fig. 9(b). shows the activity diagram after the solution of multiple start node using algorithm 1.

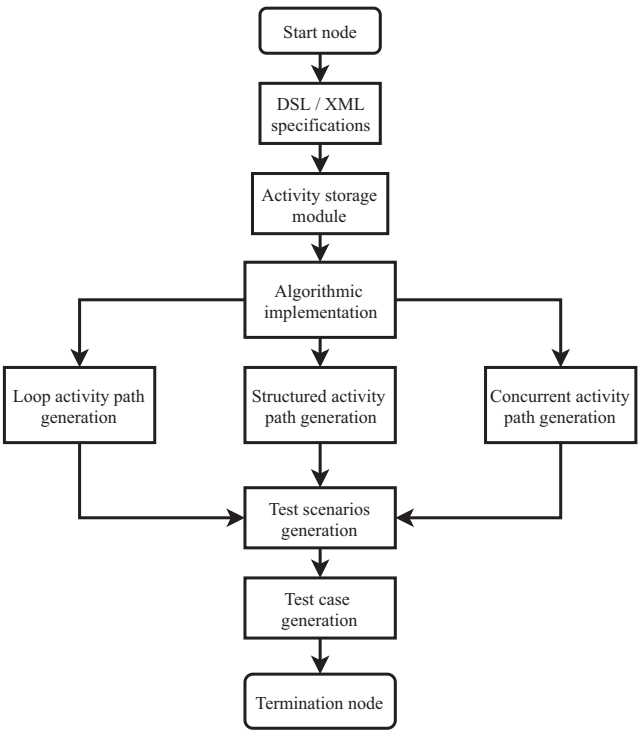


Fig. 8 Automated test case development flow

5 Automated Test Case Development Framework

The goal of scenario generation is to generate test case scripts for advance automated test case verification. The test cases scenarios is always dependent on the test case scenario generation algorithms [3, 7]. The simulation result of the generated test cases are simulated for SoC verification in its functional constraint. The automated test case development framework is shown in Fig. 8.

The XML specifications of DUV and DSL specifications of design registers are required at the start node followed by an activity storage module. The storage module replicates all the DUV signal flow activities. The algorithmic implementation exerts the rational logic on the activity diagram according to its flow. The activity flow categories are structured, loop, concurrent and multiple start node. Algorithm 1 is an appropriate solution for multiple start node problem in activity diagram, whereas algorithm 2 relevantly defines the solution to loop and concurrent activity flow. The algorithm 2 entails DFS transformation mode for transformation of structured activities into non-cyclic graphs in activity diagrams. This method generates various test path scenarios in form of activity flow. Each test path scenario represents a different test case. The inspiring case of mixed activities is shown in Fig. 9(a).

The inspiring case is incorporated in the framework to authorize the desired generation of test scenarios. The solution of activity diagram follows certain steps:

- Step I works on the loop identification. In this step flow finds while-do loop and separate it as shown in Fig. 9(a). The while-do loop box becomes the next depiction of loop.
- Step II targets to find non join concurrent activity. After finding the activity, the flow encapsulates the non join concurrent activity in a separate box as depicted in Fig. 9(a).
- In step III, the flow solves the problem of multiple start node (if exist). Algorithm 1 suitably reconstructs

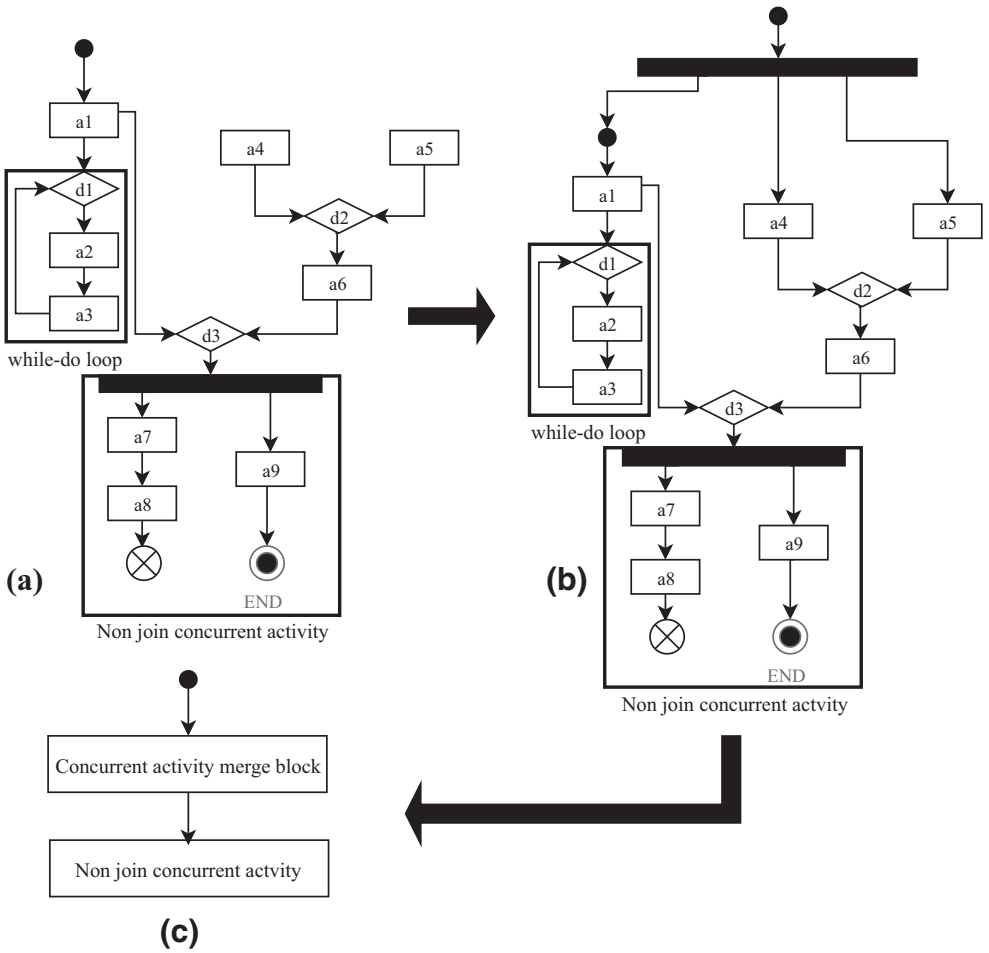


Fig. 9 (**a**) Inspiring case of mixed activities. (**b**) Solving while-do loop ad multiple start node problem. (**c**) Concurrent activity and non-join concurrent activity merge

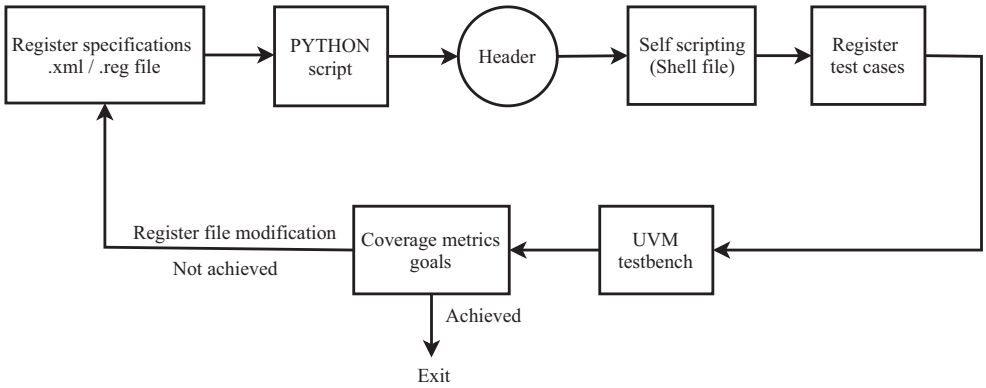


Fig. 10 Register header verification flow for generating automated test cases and attaining acceptable coverage

the multiple start node activity diagram, as depicted in Fig. 9(b).

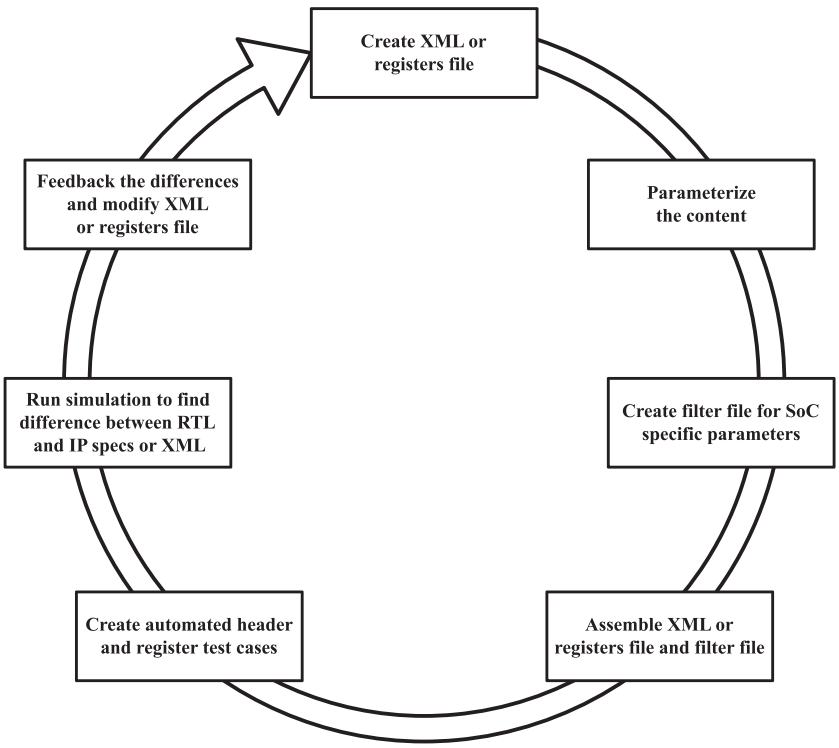
- Step IV works to merge the concurrent activities in the existing activity flow. The merger makes two modules: One is the concurrent activity merge block and other is non join concurrent activity, to create one scenario during scenario generation as shown in Fig. 9(c).

6 Automated Register Test Case Generation with Golden Header

Design register verification is also necessary to ensure the correct functionality of all registers in design. The golden header includes all the register names in an SoC, their reset value, absolute/offset address and bit field behavior. It is generated from a collection of .xml/.reg file. It is also used to generate a register model consisting of all the necessary register attributes. The work uses spreadsheet format to write XML based register specifications. The generated header is the UVM definition of register block. The self scripting shell file (.sh) generates the testcases for register verification, as shown in Fig. 10. UVM testbench simulates all the test cases. The primary objective is to gain acceptable coverage. The unaccepted coverage leads to modification of the register specification file.

The parsing of the XML file creates the register header file. The automated register header file consists of register

Fig. 11 Register header creation and verification flow



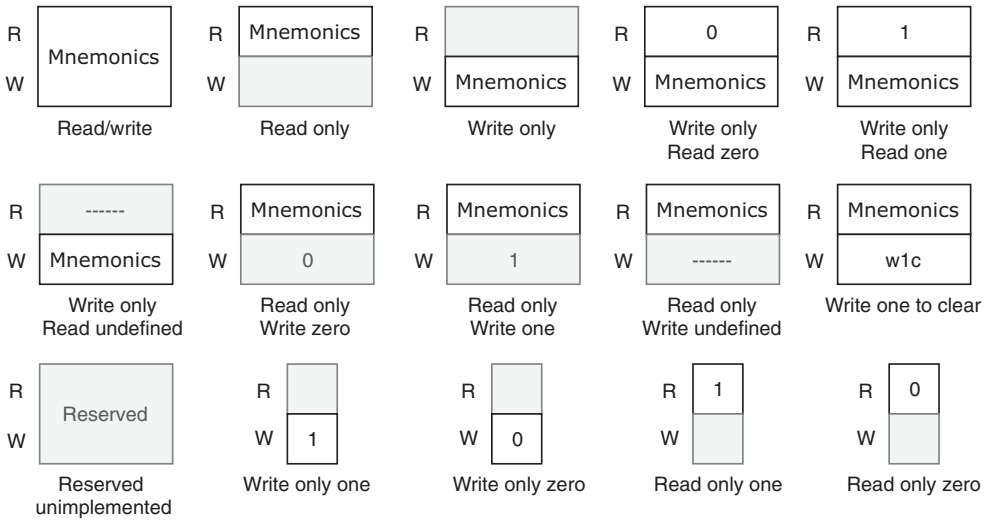


Fig. 12 Register contentions used in the proposed automated register verification model

configurations. Register model lists, create, configure, build, add map and set offset, which allows modeling of registers database. The register configuration informs about the block, address, bit width, reset value, field, bits, "register to" activity, "register from" activity and register contention type of the register. Registers may consist of depending activities during stimuli transactions. So a header file extracts "register to and register from" activities from register configuration in the header file to execute relevant tests. The single or multiple write and read test cases on dependent or independent registers are from a source register to target register/s for dependent registers. Register verification cycle starts with the creation of XML or register file. The parameterization of register content is followed by the creation of filter file for SoC specific parameters, as shown in Fig. 11. In the next phase of the cycle, XML or registers and filter files are assembled to create automated register header and register test cases. The test cases simulation finds the difference between RTL and XML specs file. The difference feedback is reported to modify the XML or register file until RTL and XML specs hit the match.

Register header verification checks the reset value of registers and access types of register bits. Test cases are created according to the reset values and access types of registers. The test cases include all conventions of registers shown in Fig. 12. Header verification process follows the following steps:-

1. `Testcases checked in at v_sc_*_reg_stim`
   `<Module>_reset_value_check.c`
   `<Module>_rmo_reg_access_check.c`
2. `Header file: reg_header.h`
3. `rmo_functions.h`
4. `csvs`

The work proposes two approaches for automated register header verification. One approach is to enable clocks only and bypass reset of the pre_main as shown in Listing 1.

The other proposed approach possesses awareness regarding various register masks. This approach dumps different masks like WORZ (write only read zero), RU (reserved unimplemented), ROO (read only one), WORO (write only read one), ROWU (read only write undefined), ROWO (read only write one), WOO (write one only), WORU (write only read undefined) denoted by mask1, mask2, mask3, mask4, mask5, mask6, mask7 and mask8 respectively. The definitions of these masks are elaborated in Fig. 12. Table 1 states the description of various register titles and their corresponding default address values. Table 1

```
#ifndef SKIP_PRE_POST_MAIN
#define pre_main( ) \
do \
{/*Enable clocks to all gated modules*/ \
ENABLE_ALL_CLK_GATING( ); \
ENABLE_CLOCK( ); \
/* Enable cache */ \
CACHE_INIT( ); \
/* Disable watchdog timer */ \
DISABLE_WDOG( ); \
/* Configure Clock Ratio */ \
CLOCK_RATIO_CONFIGURE( ); \
/* Configure chip into SOSC mode, speedup simulation */ \
CONFIGURE_SYS_CLOCK( ); \
/* Configure for clk_on options */ \
CONFIGURE_CLK_ON( ); \
/* Clear SRTC interrupt */ \
CLEAR_SRTC_INT_AFTER_POR( ); \
/* Relocate the vector table */ \
RELOCATE_INTERRUPT_VECTOR_TABLE( ); \
/* Check if wakeup from LLMU wakeup source */
srs_hold = REG8(RCM_RBASE+0x08); \
/*Clear ACKISO */ \
CLEAR_ALL_PENDING_SOURCES( ); \
ENABLE_ALL_INTERRUPT_SOURCE( ); \
SET_ATX_CLK( ); \
/* Increment reset counter */ \
++capi_reset_counter ; \
} while(0)
#else
#define pre_main( ) \
do\
{ /* enable clocks to al gated modules */ \
ENABLE_ALL_CLK_GATING( ); \
/* Clear SRTC interrupt */
CLEAR_SRTC_INT_AFTER_POR( ); \
}while(0)
#endif // SKIP_PRE_POST_MAIN
#define post_main( ) do{} while(0)
```

Listing 1 Code representation to only enable clocks and bypass reset of the pre_main

Table 1 Override entries on .reg register specification file

Register Title	Definition	Default Value
DONT_READ	Utility shall not dump read enable RE32/16/8 for such registers in .c testcase	No
DONT_WRITE	Utility shall not dump write enable WE32/16/8 for such registers eg: RO bits which generate xfr_error on write (reg: TKT218796)	No
RESET_VALUE_MASK	While checking reset value, don't check the bits whose mask is '0'. By default only those bits will be compared which have reset_value mask as '1'.	0xFFFFFFFF
RESET_VALUE_OVERRIDE	PAD settings or flash record changes the reset value, then use this override which will change the expected value in reset checking(RE)	No
WRITE_1_MASK	If ONE is not to be written to few bits of registers (while performing write_all_ones task). Then a mask like 0x00000001 means that 0th bit is not to be written '1' .	0x00000000
WRITE_0_MASK	If ZERO is not to be written to few bits of registers (while performing write_all_zeros task) then the value of mask should be set for those bits.	0x00000000
READ_1_MASK	If ONE is not to be checked on few bits of registers(while performing check ones task).Then, don't check the bits which are marked '1'.	0x00000000
READ_0_MASK	If ZERO is not to be checked on few bits of registers (while performing check_zeros task). Then, don't check the bits which are marked '1'.	0x00000000
ENTER_CONFIG_MODE	Few registers are writable/readable only after writing a sequence of registers. Add the sequence in this function. Auto generated script will append this task prior accessing the register.	–
EXIT_CONFIG_MODE	Auto generated script will append this task after accessing the register to restore the system settings done in ENTER_CONFIG_MODE.	–
INTERMEDIATE_CONFIG_MODE	If any sequence is needed between the resister ie after writing 0's and before writing 1's any sequence is needed then add the sequence in function Auto generated script will append this function between 0's and 1's writing register.Ex write Once register TKT342378	–
Applicable for Verification	If it is "YES", then only values of masks & overrides of previous columns of csv will be appended on existing .c cases else no appending will take place .This field will be set to yes by Verif folks	Yes
Applicable for Validation	If it is "YES", then only values of masks & overrides of previous columns of csv will be appended on existing .c cases else no appending will take place . This field will be set to yes by valid folks	Yes
Testcase_Covering_the_discounted_Feature in verification	Here, a testcase name has to be added which is implemented at verif's end which covers all bits that were masked & not checked by parsing of csv.	Null
Testcase_Covering_the_discounted_Feature in validation	Here, a testcase name has to be added which is implemented at validation's end which covers all bits that were masked & not checked by parsing of csv.	Null
ENTER_CONFIG_MODE_RESET	Few registers are readable only after writing a sequence of registers. Add the sequence in this function. The auto-generated script will append this task in Reset value check prior reading the reset value of the register.	–
EXIT_CONFIG_MODE_RESET	Auto generated script will append this task in reset value check testcase after Reading the register to restore the system settings done in ENTER_CONFIG_MODE_RESET.	–
COMMENTS	Relevant comment for the entry in any column of CSVs.	No

states the override value on CSV entries. To commence the pertinent verification on registers masks, it is essential to override the CSV entries of respective masks, as shown in Table 1. These entries are defined in a function called rmo (register mask override) as depicted in Listing 2.

The entries in function rmo are the detailed invasion of register masks, register initial values and override values. Function rmo rationally obtains the correct output with appropriate mask bits. The separate mask bits are prepared to execute AND/OR operation of mask bits with initial value and desired value. The mask bit size is the same as output bit size. Listing 2 shows a 32 bit rmo function declaration. Mask1 is WORZ (32 bits), mask2 is RU for bits reservation (place all reserved bits from 1s and mask bits from 0s), mask3 is write 1 mask (After logic AND/OR operation, mask3 generates all 1s), similarly, all masks works on predefined logic operation to acquire desirable result. A 32 bit stream of bits_which_will _read_zero generates is the result of AND between initial value, mask6 and mask7 (mask6 and mask7 are write one mask), So the resultant bits generates 0s for read zero scenario and rest all 1s. The flag is raised for mask bits. The rmo function generates the register header file corresponding to reg_0 is shown in Listing 3.

The generated header file is the captivating response of rmo function. The header consists of all mandatory register fields including the entail mask parameters with their respective memory addresses. It comprises of register size, type, reset value, and all masks contentions to be verified. The header file (register database) is the combination of all SoC registers and their respective fields. The automated register header enhances the productivity boost in verification. Simulation of design takes a little more processing time in the start due to the processing involved in header creation. However, later it significantly vanishes the manual effort to create the register database. Post header

```
// This file exclusively contains functions only for RMO usage & is included in environment.h
typedef uint32_t reg_addr_t;
typedef uint32_t reg_32data_t;
typedef uint16_t reg_16data_t;
typedef uint8_t reg_8data_t;
//
// ************32 bit functions ******************//
reg_32data_t write_one_value32(reg_32data_t output , reg_32data_t initial_value , reg_32data_t Mask1,
        reg_32data_t Mask2, reg_32data_t Mask3)
{output = ( initial_value | (~Mask2)); // Mask 2= Reserved bits, write initial value (already written)
        on all the reserved bits
output = output & (~Mask1) & (~Mask3);//Mask 1 is ROWZ. write zero only and Mask 3 is ROO (NOTE:
        write_one_mask value from CSV file for special scenarios)
SHOW32("output********","output=",output);
return output;
}
reg_32data_t expect_one_value32(reg_32data_t output ,reg_32data_t initial value, reg_32data_t Mask1,
        reg_32data_t Mask2, reg_32data_t Mask3, reg_32data_t Mask4, reg_32data_t Mask5,
        reg_32data_Mask6, reg_32data_t Mask7, reg_32data_t Mask8, reg_32data_t Mask9)
{reg_32data_t Mask,bits_which_will_read_zero;
Mask = (Mask1 | Mask2 | Mask5 ) & initial_value ;
bits_which_will_read_zero = initial_value & (~Mask6) & (~Mask7)
output = (Mask| Mask4| Mask8| Mask9 | bits_which_will_read_zero ) & (~Mask3);
SHOW32("output********","output=",output);
return output;}
reg_32data_t mask_one_value32(reg_32data_t output , reg_32data_t Mask1, reg_32data_t Mask2,
        reg_32data_t Mask3, reg_32data_t Mask4)
{output = (Mask1 | Mask2 | Mask3 | Mask4);
SHOW32("output********","output=",output);
return output;
}
```

Listing 2 Register mask override (rmo) function

```
#define REG_0_BASEADDRESS                0x40160000
/* Register definitions */
/* Module Configuration Register */
#define REG_0_MCR_REGOFFSET              0x00000000
#define aREG_0_MCR                       (*(vuint32_t*)(REG_0_BASEADDRESS+0x00000000))
#define iREG_0_MCR                       (*(vuint32_t*)(REG_0_BASEADDRESS+0x00000000))
#define REG_0_MCR_MCR                    (*(vuint32_t*)(REG_0_BASEADDRESS+
                                         REG_0_MCR_REGOFFSET))
#define REG_0_X_MCR(x)          (*(vuint32_t*)(REG_0_BASEADDRESS+REG_0_MCR_REGOFFSET))
                                         +((x)*0x00001000)
#define REG_0_MCR_REGSIZE                32
#define REG_0_MCR_REGTYPE                vuint32_t
#define REG_0_MCR_RESET_VALUE            0xD890000F
#define REG_0_MCR_BITFIELD_MASK          0xFBB3BB7F
#define REG_0_MCR_ROMASK                 0x09100000
#define REG_0_MCR_ROOMASK                0x00000000
#define REG_0_MCR_ROWOMASK               0x00000000
#define REG_0_MCR_ROWUMASK               0x00000000
#define REG_0_MCR_ROWZMASB               0x00000000
#define REG_0_MCR_ROZMASK                0x00004480
#define REG_0_MCR_RUMASK                 0x00000000
#define REG_0_MCR_RWMASK                 0xF6EFBB7F
#define REG_0_MCR_W1MASK                 0x00000000
#define REG_0_MCR_W0MASK                 0x00000000
#define REG_0_MCR_WOMASK                 0x00000000
#define REG_0_MCR_WOOMASK                0x00000000
#define REG_0_MCR_WOROMASK               0x00000000
#define REG_0_MCR_WORUMASK               0x00000000
#define REG_0_MCR_WORZMASK               0x00000000
#define REG_0_MCR_WOZMASK                0x00000000
#define REG_0_MCR_WONCEMASK              0x00000000
#define REG_0_MCR_UNUSED_MASK            0x044C4480
#define REG_0_MCR_MAXMB_LSB              0    /*Number Of The Last Message Buffer */
#define REG_0_MCR_MAXMB_MSB              6
#define REG_0_MCR_MAXMB_MASK             0x0000007F
#define REG_0_MCR_MAXMB                  ((REG_0_MCR_MAXMB_MASK)<<(
                                         REG_0_MCR_MAXMB_LSB))
#define REG_0_MCR_MAXMB_VALUE(x)         (((x)&0x0000007F)<< 0)
#define REG_0_MCR_IDAM_LSB               8    /*ID Acceptance Mode */
#define REG_0_MCR_IDAM_MSB               9
#define REG_0_MCR_IDAM_MASK              0x00000300
```

Listing 3 Created register header file or automated register database using rmo function

creation, the testbench architecture consumes lesser simulation time due to optimized code and separate database as displayed in Fig. 1. The header file acts as the predominate input to self scripting shell file for register verification as depicted in Fig. 10. The test automation shell script generates numerous C test cases for SoC register verification.

The C test cases take a testbench run to accumulate the feedback of bug resistant algorithm in the form of functional coverage. The generated test cases trigger various general combinations of single write or read to multiple write or read on 8 bits, 16 bit, and 32 bit SoC registers. This automation relinquishes manual intervention for writing test cases for design verification. Listing 4 exhibits the auto-generated C tests. The test cases scripts the SoC registers for writing all 0s and reading, writing all 1s and reading test cases, taking the various single register and multiple register read-write concerns. Register test cases possess reset, timer, flag, control and mask register parameters for logical interpretation of single or multiple write-read combinations.

7 Results and Discussions

7.1 Functional Coverage and Cross-Functional Coverage

The work studies a five power domain, 316 registers SoC to validate the results. The SoC validation uses two

```
#ifdef VAL_CODE
#include <reg_header.h>
#include " verify .h"
int reg_0_reg_reset ( )
#else /* VAL CODE */
#include <UVM_environment.h>
#include <reg_header.h>
#include <rmo_functions.h>
int main( )
#endif /* VAL CODE */
{
INFO ( " reg0_reg_reset_stimulus ** Testcase starts ***);
INFO ( "** Standard Stimulus REG reset" , "Check register reset of reg_0 DIGITAL memory
address **" );

REM32(REG_0_MCR,REG_0_MCR_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_CTRLA,REG_0_CTRLA_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_SOCTIMER,REG_0_SOCTIMER_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_TRANS,REG_0_TRANS_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_RXSOCMASK,REG_0_RXSOCMASK_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_RXMASK14,REG_0_RXMASK14_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_RXMASK15,REG_0_RXMASK15_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_ENTRY,REG_0_ENTRY_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_EXIT,REG_0_EXIT_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_SOCMASK1,REG_0_SOCMASK1_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_SOCMASK2,REG_0_SOCMASK2_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_SOCFLAG1,REG_0_SOCFLAG1_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_SOCFLAG2,REG_0_SOCFLAG2_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_CTRLB,REG_0_CTRLB_RESET_VALUE, 0xFFFFFFFF);
REM32(REG_0_ECR,REG_0_ECR_RESET_VALUE, 0xFFFFFFFF);}
{INFO("reg0_rmo_reg_access_stimulus"," testcase starts "):
INFO("Standard stimulus REG access","check register access of REG_0 address);
uint32_t initial_32value ,backup_32value,write_one_32value, write_zero_32value ,
expect_one_32value,expect_zero_32value ,mask_one_32value,mask_zero_32value;
uint16_t initial_16value ,backup_16value,write_one_16value, write_zero_16value ,
expect_one_16value,expect_zero_16value ,mask_one_16value,mask_zero_16value;
uint8_t initial_8value ,backup_8value,write_one_8value , write_zero_8value ,
expect_one_8value,expect_zero_8value ,mask_one_8value,mask_zero_8value;

INFO("//Writing all 0s on REGISTER","REG_0_MCR register//");
R32((*( vuint8_t *)&REG_0_MCR).initial_32value);
SHOW32("initial_32value"," initial_32value =", initial_32value );
backup_32value=in1tial_32value ;
write_zero_32value=wrlte_zero value32( write_zero_32value , initial_32value , REG_0_MCR_RWMASK,
REG_0_MCR_WOOMASK,0,REG_0_MCR RUMASK, REG_0_MCR_WlCMASK):
expect_zero_32value=expect_zero_value32(expect_zero_32value , initial_32value ,REG_0_MCR_RWMASK,
REG_0_MCR_ROMASK,0,REG_0_MCR_RUMASK,REG_0_MCR_ROOMASK,REG_0_MCR_WOROMASK,
REG_0_MCR_W1CMASK,REG_0_MCR_WOOMASK,0);
mask_zero_32value=mask_zero_value32(mask_zero_32value,0,REG_0_MCR_WOMASK,
REG_0_MCR_ROWUMASK,REG_0_MCR_WORUMASK);
WREM32((*(vuint8_t*)&REG_0_MCR), write_zero_32value,expect_zero_32value,mask_zero_32value);

INFO ("//Writing all 1s on the", "REG_0_MCR register//"):
R32((*( vuint8_t*)&REG_0_MCR),initial_32value);
SHOW32("initial_32value"," initial_32value =", initial_32value );
write_one_32value=wrlte_one_value32(write_one_32value,REG_0_MCR_ROWZMASK,
REG_0_MCR_RUMASK, 0);
expect_one_32value=expect_one_value32(expect_one_32value, lnitial_32value ,REG_0_MCR_ROMASK,
REG_0_MCR_RUMASK,0,
REG_0_MCR_ROOMASK,REG_0_MCR_ROWZMASK,REG_0_MCR_W1CMASK,
REG_0_MCR_W0RZMASK,REG_0_MCR_RWMASK,REG_0_MCR_WOROMASK);
mask_one_32value=mask_one_value32(mask_one_32value,0,REG_0_MCR_WOMASK,
REG_0_MCR_ROWUMASK,REG_0_MCR_WORUMASK);
WREM32((*(vuint8_t*)&REG_0_MCR),write_one_32value,expect_one_32value,mask_one_32value);
WREM32((*(vuint8_t*)&REG_0_MCR),backup_32value);}
```

Listing 4 Generated register test cases from proposed methodology

cases: one uses UVM constraint random sequences for SoC verification [10, 20], Other uses automated test intent. The results are attained by running various sequences with different tests with different seeds. In both cases, SoC cover groups are simulated to gain persistent coverage data. We acquire 93.25% coverage from the cast using UVM constraint random sequence, as shown in Fig. 13. Fig. 13 depicts that it took 79000 (bin auto[0] + bin auto[1]) transactions to attain 100% coverage for individual coverpoints in a good perspective. Unfortunately, chunk size vs. total size cross coverpoints has only reached 29.22% coverage. In a deeper view, it can be seen from Fig. 13 that some of the CROSS coverpoint combinations hit several times (3, 3, 8, 1, 1), where many of them remain missed (0, 0). If we are limited to use only UVM constraint random transactions for stimulus generation. The solution is to add more constraints to the descriptor

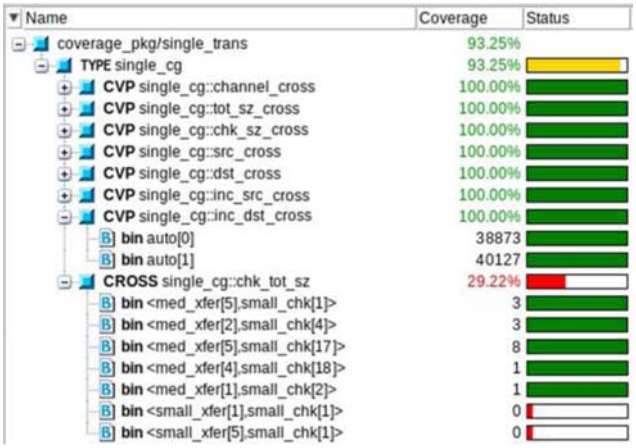


Fig. 13 Functional and cross-functional coverage of UVM constraint random architecture

object or to create a separate sequence type to constrain the generated transactions to hit the missing scenarios. These solutions consume a considerable amount of time for coverage analysis and constraint modification to hit the missing scenarios. After alteration, the simulation has to run again. However, we have already wasted 79000 transactions in the previous run. The simulation cycle is a non-renewable resource; once it is consumed, we cannot get it back. So it is imperative to maximize the use of each simulation cycle. UVM constraint random is good, but some modifications are necessary to meet the standard objectivity.

The work import the UVM transactions and covergroups to test intent architecture to make UVM environment more efficient. It improves the coverage associativity of SoC verification using test intent model. The purpose is to get rid of shortcomings due to UVM constraint random stimulus [20]. Test intent defines the coverage in an upfront manner. It prioritizes the test cases explicitly to meet the coverage goal. By analyzing the scenarios in graphical format, test intent identifies the key stimulus values, cross combinations, and sequences to guarantee the perfect coverage goal. The generated test cases automatically suppress the redundant values in the stimulus. This enables the test intent model to achieve the verification goal in comparatively lesser time. It also verifies more graphical scenarios. The coverage using test intent architecture is shown in Fig. 14.

The work acquires 100% coverage in individual and cross combinations of coverpoints. The proposed approach completes the DUV verification in 7736 transactions. The CROSS coverpoints are hit only one time; it reduces the redundancy in the simulation cycle. The bug resistant module assures the absence of bugs to enhance overall verification productivity.

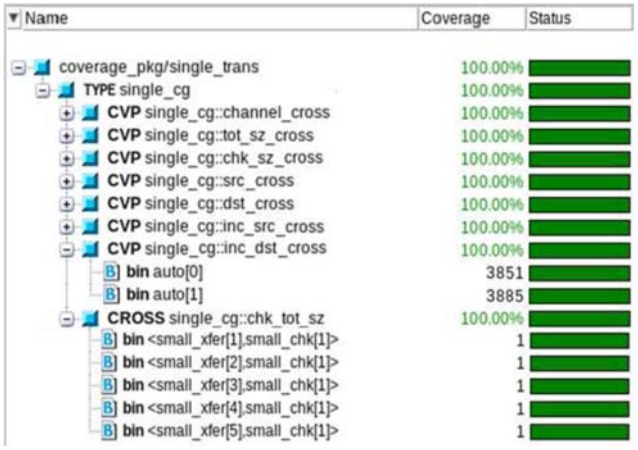
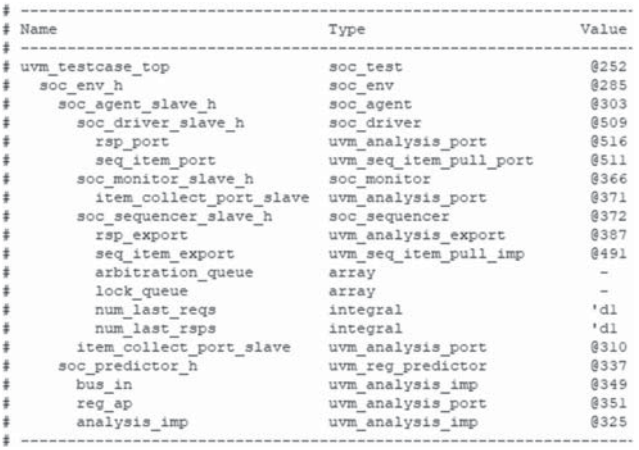


Fig. 14 Functional and cross-functional coverage of proposed test intent architecture



```
# ----------------------------------------------------------------
# Name                          Type                    Value
# ----------------------------------------------------------------
# uvm_testcase_top              soc_test                @252
#   soc_env_h                   soc_env                 @285
#     soc_agent_slave_h         soc_agent               @303
#       soc_driver_slave_h      soc_driver              @509
#         rsp_port              uvm_analysis_port       @516
#         seq_item_port         uvm_seq_item_pull_port  @511
#       soc_monitor_slave_h     soc_monitor             @366
#         item_collect_port_slave uvm_analysis_port     @371
#       soc_sequencer_slave_h   soc_sequencer           @372
#         rsp_export            uvm_analysis_export     @387
#         seq_item_export       uvm_seq_item_pull_imp   @491
#         arbitration_queue     array                   -
#         lock_queue            array                   -
#         num_last_reqs         integral                'd1
#         num_last_rsps         integral                'd1
#       item_collect_port_slave uvm_analysis_port       @310
#     soc_predictor_h           uvm_reg_predictor       @337
#       bus_in                  uvm_analysis_imp        @349
#       reg_ap                  uvm_analysis_port       @351
#       analysis_imp            uvm_analysis_imp        @325
# ----------------------------------------------------------------
```

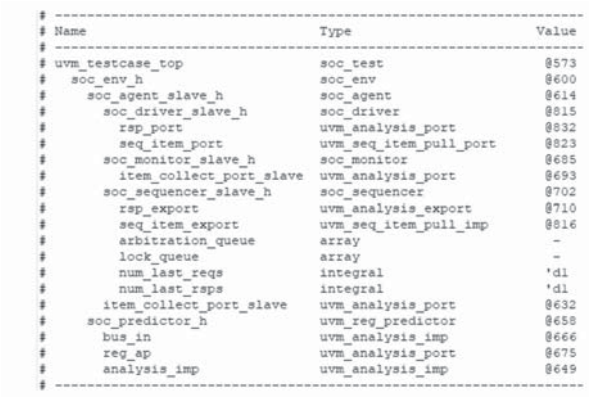
Fig. 16 Simulation handle values of proposed test intent architecture

7.2 Module Handle Comparison

The module handles well inform the prior access of module. The handles of modules are read in order to capture their values. The handle value of each architecture module identifies the module code entry scenario. Fig. 15 shows the typical conventional UVM constraint random verification handle values for environment modules.

Similarly, Fig. 16 extracts the better handle value of modules using the proposed test intent architecture. In spite of the extra processing during the execution of the bug resistant algorithm, the proposed test intent is way faster than conventional constraint random UVM architecture. It also eliminates the need of manual intervention to write code for register database and test cases. Although the test intent needs an in-depth understanding of design specifications, it gives proof and trace of the bug in the tree-level graph scenario.

Table 2 depicts the handle values of concerning modules. The analysis of Table 2 shows the exact handle improvement after 7736 iterations. The final reallocation illustrates the improvement for scenario model database output port, test automation block output port, UVM testbench environment, environment agent, environment sequencer, environment driver and environment monitor by handle value of 176, 193, 321, 293, 330, 306 and 319 respectively. The average line fit handle improvement reaches 276 after 7736 iterations. The coverage in the conventional verification methods is entirely data-centric. It extracts the monitor input covergroup parameters and constraints applied on actions. The proposed works adds more coverage type to the existing ones, such as action coverage, scenario coverage, datapath coverage and source coverage. The system level scenarios and model is made compatible enough to acquire the legal scenarios.



```
# ----------------------------------------------------------------
# Name                          Type                    Value
# ----------------------------------------------------------------
# uvm_testcase_top              soc_test                @573
#   soc_env_h                   soc_env                 @600
#     soc_agent_slave_h         soc_agent               @614
#       soc_driver_slave_h      soc_driver              @815
#         rsp_port              uvm_analysis_port       @832
#         seq_item_port         uvm_seq_item_pull_port  @823
#       soc_monitor_slave_h     soc_monitor             @685
#         item_collect_port_slave uvm_analysis_port     @693
#       soc_sequencer_slave_h   soc_sequencer           @702
#         rsp_export            uvm_analysis_export     @710
#         seq_item_export       uvm_seq_item_pull_imp   @816
#         arbitration_queue     array                   -
#         lock_queue            array                   -
#         num_last_reqs         integral                'd1
#         num_last_rsps         integral                'd1
#       item_collect_port_slave uvm_analysis_port       @632
#     soc_predictor_h           uvm_reg_predictor       @658
#       bus_in                  uvm_analysis_imp        @666
#       reg_ap                  uvm_analysis_port       @675
#       analysis_imp            uvm_analysis_imp        @649
# ----------------------------------------------------------------
```

Fig. 15 Simulation handle values of UVM constraint random architecture

Table 2 Module handle comparison between UVM constraint random and proposed test intent architecture

Module name	Handle value of UVM constraint random stimulus	Handle value of proposed architecture	Improve handle value
Scenario model database output port	369	193	176
Test automation block output port	406	213	193
UVM testbench environment	573	252	321
Environment agent	598	305	293
Environment sequencer	702	372	330
Environment driver	815	509	306
Environment monitor	685	366	319

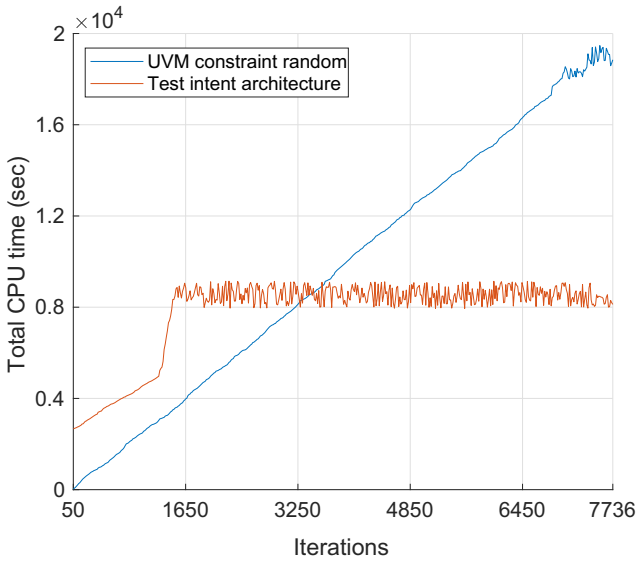


Fig. 17 Total CPU time comparison between UVM constraint random and proposed test intent architecture

7.3 CPU Time, Coverage and Throughput Comparison

The total CPU time computes the total processing time for execution of instructions in a computer program. CPU time is denoted by seconds or clock ticks. The CPU time comparison study is illustrated in Fig. 17. The experimentation assures the inactivity of all non-related background running computer applications to examine the CPU time. The work uses core i7 2.4 GHz processor on scientific LINUX operating system. Mentor graphics Questasim 10.5a is utilized to depict the results. Fig. 17 shows the comparison of conventional UVM constraint random architecture and test intent architecture.

Initially, Fig. 17 shows that the UVM constraint random architecture consumes lesser CPU time as compared to test intent architecture. The test intent architecture requires extra CPU/process time to automate the database and design test cases while the UVM constraint random architecture directly starts DUV verification through manual test cases. UVM constraint random and test intent architectures consume 23 sec and 3188 sec for the first 50 iterations. The huge difference in the total CPU time in Fig. 17, is for the excess processing involved for automatic creation of the register-coverage database and test cases. The CPU time of proposed test intent architecture becomes constant in a range later on to improve the predictability of execution scenario. The UVM constraint random architecture possesses continuous increment in CPU time. Although after 7185 iterations the CPU time escalation halts in a wide range due to cyclic randomization. Cyclic randomization excludes the design parameters which are already covered. The UVM constraint randomization assures the non-repetitive transactions. After certain iterations, the processing of the program instructions becomes faster because of lesser data left to execute and the (Verilog Procedural Interface) VPI subroutine simulator path also activates for faster execution. VPI routine simulator path works for lesser memory; It stores the next instruction to be executed for the processor to improve the CPU time. Although this also boosts CPU processing for small databases, it works a little fine rather than not using it. The command in the program script decides the activation of VPI path employing certain algorithms. The algorithms work in a precise manner to calculate the number of program instructions left along with their processing time estimation. VPI routine path undoubtedly makes the coverage faster but at the cost of functional coverage. The direct intervention of transactions to DUV makes the verification error-prone due to the absence of bug removal methods. The VPI method is direct so it works without self-checking assertions.

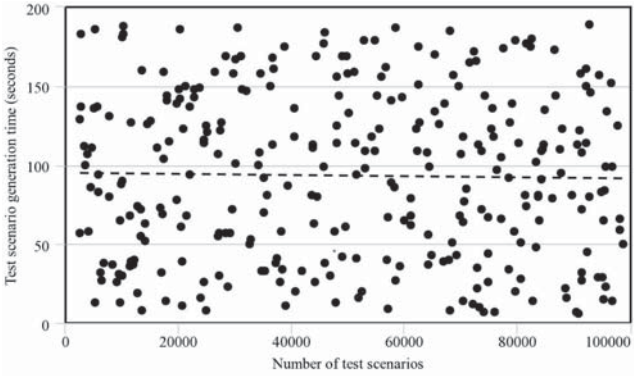
Table 3 elucidates the comparison between the proposed approach and state of the art methodologies [3, 8, 21]. The comparison results shows that the proposed methodology consumes the least CPU processing time and produces maximum throughput in producing design scenarios. The CWL (component wrapper language (CWL)) TLM verification [8] consumes the highest processing and least throughput for scenario conversion because it doesn't work on separate pre-processed database. UVM constraint-random method [10] without processed database and scenarios automates

Table 3 Comparison of test intent with conventional verification design methodologies

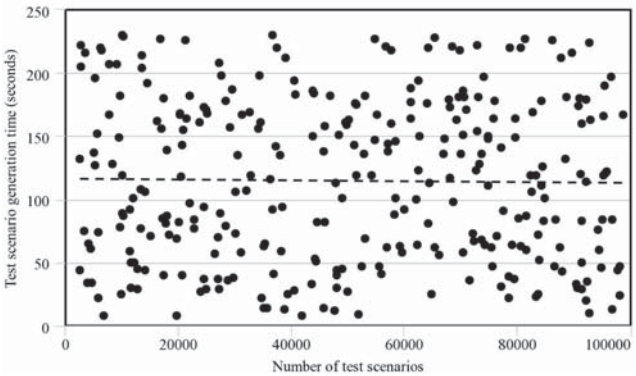
Approach	Iterations	Register test scenarios	No. of design test scenarios	CPU time	Coverage	CPU Throughput $\times 10^{-4}$
CWL transaction-level [8]	82817	471	3135	2.21×10^4sec	99.75%	1,631.67
UVM constraint random [20]	79000	376	2845	1.97×10^4sec	93.25%	1635.02
Scenario model [3]	18346	469	3283	1.34×10^4sec	100%	2738.68
Arbitrary-Distribution Constrained Random [21]	10158	418	3118	$.96\times10^4$sec	100%	3683.33
Proposed test intent	7736	423	3245	$.83\times10^4$sec	100%	4419.27

the tests with less functional coverage due to its inability to cover wider register scenarios. The Scenario model [3] is generating more register and design test scenarios than proposed architecture but scenario model doesn't occupy any provision for bug removal, so it intentionally write/read on masked bits of register and SoC inputs to cover more, but unauthorized scenarios. The arbitrary distribution constrained random [21] uses density estimation techniques and RSSDE algorithm for arbitrary distribution, tree guided pattern generation and error accumulation to tree. The arbitrary distribution uses probabilistic methods for error accumulation in scenario generation, which leaves corner cases at lower confidence level. The proposed approach significantly improves the CPU time, functional coverage and throughput to accomplish the design verification in minimum iterations as shown in Table 3.

The proposed test intent architecture promotes the automated databases of DUV registers and input coverage variables to ease the instruction fetch. It evolves with a faster CPU processing time compared to the conventional UVM constraint random method. The constant range variation in the total CPU time is a perfect verdict to prove the capability of test intent architecture over "UVM constraint random architecture - the one of the fastest existing verification method" [21]. Fig. 18 compares the proposed test intent architecture with the arbitrary-distribution constrained random architecture. Fig. 18a illustrates the test scenario generation time of proposed test intent. The proposed test intent takes 198 seconds run time for 10^5 pattern generation with a mean best fit of 98.96 seconds, whereas the arbitrary-distribution constrained random architecture shown in Fig. 18b takes 232 seconds run time for test pattern generation with a mean fit of 117.65 seconds for the same number of patterns. The method proposed by arbitrary-distribution constrained random architecture is based on probabilistic analysis for generation of trees and error evaluation with confidence level between 80%-95%. The probabilistic methods have corner case verification issues due to confusing probability measures. If the probability is nearly 0.5 in corner cases, it may take the wrong tree scenario test pattern or skip the pattern of scenario. Although the chances of pattern skipping is very low above 90% confidence level for huge SoCs, there are enough provisions for skipping scenario. The probabilities nearly 0.5 counters an algorithm for final path determination to increase the confidence level, also increases the processing time during pattern generation. The proposed methodology uses bug resistant weight-based automation to remove subroutine calls during pattern generation. The algorithm is designed to counter corner cases and bugs. The bugs mean to write or read on the registers or memory according to its contentions like it removes the test scenarios which are trying write/read on



(a) Test scenario generation time scatter graph for proposed architecture



(b) Test scenario generation time scatter graph for arbitrary-distribution constrained random architecture [21]

Fig. 18 Test scenario generation time comparison between proposed and arbitrary-distribution constrained random architecture [21]

mask bits, write on read-only bits, read on write-only bits and many constraints according to its contentions. The bug eradication refines the flow for practical scenarios only to lessen the load during pattern generation. Weight analysis is based on analysis of data/module type, contention, specification and projection, to avoid any subroutine processing during pattern generation, this leads the test intent to generate test patterns in lesser time as compared to well-known approaches.

8 Conclusion

In this paper, an automated test case generation with no repetitive scenario trace approach is presented. The method eliminates the requirement to write the test cases manually. The work also shows the automatic register database creation of all SoC register contentions. The functional coverage comes out to be 100% for the model using the proposed bug resistant algorithm. DSL description takes a better advantage on component wrapper language

(CWL) description. All the present bugs are detected before 133 patterns. The cyclic scenario-based transaction structure consumes lesser CPU/process time as compared to the primitive UVM constraint random verification. The work provides early access to the verification environment depicted by comparing handle values. Although the proposed architecture is a success, at the start, it takes high processing time; this makes it unsuitable for verification of smaller designs. The design needs to be more significant enough to compensate its initial high processing constraints in the later stages of verification. As future work, we would like to extend the present model by incorporating design deadlock scenarios generated by inappropriate stimuli transactions during verification.

Acknowledgements This work is supported by the Ministry of Electronics and Information Technology (Meity), Government of India.

References

1. Santarini M (2005) Asic prototyping: Make versus buy. 50(11):30–40
2. Bich Ngoc Do T, Kitamura T, Nguyen VT, Hatayama G, Sakuragi S, Ohsaki H (2013) Constructing test cases for n-wise testing from tree-based test models. In: Proceedings of the fourth symposium on information and communication technology, SoICT '13. ACM, New York, pp 275–284
3. Fathy K, Salah K. (2016) An efficient scenario based testing methodology using uvm. In : Proceedings of 17th international workshop on microprocessor and soc test and verification (MTV), pp 57–60
4. Sharma G, Bhargava L, Kumar V (2018) Automated coverage register access technology on uvm framework for advanced verification. In : Proceedings of IEEE international symposium on circuits and systems (ISCAS), pp 1–4
5. Kirchsteiger CM, Trummer C, Steger C, Weiss R, Pistauer M (2008) Automatic verification plan generation to speed up soc verification. In : Proceedings of IEEE nordic circuits and systems conference (NORCAS) and international symposium of system-on-chip (SoC) (NORCHIP), pp 33–36
6. Haedicke F, Le HM, Große D, Drechsler R (2012) Crave: an advanced constrained random verification environment for systemc. In : Proceedings of International Symposium on System on Chip (SoC), pp 1–7
7. Naik K, Sarikaya B (1993) Test case verification by model checking. Form Methods in Syst Des 2(3):277–321
8. Ara K, Suzuki K (2003) A proposal for transaction-level verification with component wrapper language. In : Proceedings of design, automation and test in europe conference and exhibition, pp 82–87 suppl.
9. Auri MR, Vincenzi TB, de Oliveira DG, de Souza SRS, Maldonado JC (2016) The complementary aspect of automatically and manually generated test case sets. In: Proceedins of the 7th international workshop on automating test case design, selection, and evaluation, A-TEST 2016. ACM, New York, pp 23–30
10. Iliuţ I, Ţepuş C (2014) Constraint random stimuli and functional coverage on mixed signal verification. In: Proceedings of international semiconductor conference (CAS), pp 237–240
11. Ara K, Suzuki K (2005) Fine-grained transaction-level verification: using a variable transactor for improved coverage at the signal level. IEEE Trans Comput Aided Design Integr Circuits Syst 24(8):1234–1240
12. Cieplucha M (2019) Metric-driven verification methodology with regression management. J Electron Test 35(1):101–110
13. Yurov LV (2015) Quality assessment of verification methodologies and verification procedures. Meas Tech 58(1):38–45
14. Wakabayashi K, Okamoto T (2000) C-based soc design flow and eda tools: an asic and system vendor perspective. IEEE Trans Comput Aided Design Integr Circuits Syst 19(12):1507–1522
15. Elakkiya C, Murty NS, Babu C, Jalan G (2017) Functional coverage - driven uvm based jtag verification. In : Proc of IEEE international conference on computational intelligence and computing research (ICCIC), pp 1–7
16. Floyd. RW (1993) Assigning Meanings to Programs. Springer, Dordrecht, pp 65–81
17. El-Ashry S, Salah K (2015) A functional coverage approach for direct testing: an industrial ip as a case study. In : Proc. of IEEE EUROCON 2015 - international conference on computer as a tool (EUROCON), pp 1–6
18. Marchetto A, Islam MM, Asghar W, Susi A, Scanniello G (2016) A multi-objective technique to prioritize test cases. IEEE Trans Softw Eng 42(10):918–940
19. Tekcan T, Zlokolica V, Pekovic V, Teslic N, Gunduzalp M (2012) User-driven automatic test-case generation for dtv/stb reliable functional verification. IEEE Trans Consum Electron 58(2):587–595
20. Plaza SM, Markov IL, Bertacco V (2008) Random stimulus generation using entropy and xor constraints. In : Proceedings of Design Automation and Test in Europe, pp 664–669
21. Wu B, Yang C, Huang C (2014) A high-throughput and arbitrary-distribution pattern generator for the constrained random verification. IEEE Trans Comput Aided Design Integr Circuits Syst 33(1):139–152

Gaurav Sharma is a Ph.D. Scholar in Department of Electronics & Communication Engineering, Malaviya National Institute of Technology, Jaipur, India. His research interests include UVM verification, dynamical systems, formal verification and memristor devices. Gaurav Sharma has received his M.tech in VLSI design from Malaviya National Institute of Technology Jaipur.

Lava Bhargava is Professor in Department of Electronics & Communication Engineering Malaviya National Institute of Technology, Jaipur, India. His research interests include verification, system level design and modeling, VLSI testing & testability and formal verification. Lava Bhargava is Ph.D. from Indian Institute of Technology Delhi having 30 years of research experience.

Vinod Kumar is design & verification engineer at NXP Semiconductors, Noida, India. He has 10 years of experience as a verification engineer. He is a certified verification evangelist. His areas of interest are verification of complex designs using System Verilog and UVM, SoC Verification, designing embedded systems. He has highly explored coding, debugging skills and worked on numerous design verification projects.