Graphical Topology Info Structure for Constrained Random Verification in SoC/Subsystem Tests

Evean Qin, Richard Bell, David Chen
Advanced Micro Devices, Inc.
1 Commerce Valley Dr. East
Markham, ON Canada L3T 7X6

***Abstract* - Constrained random verification (CRV) has been widely adopted by the design verification industries. Especially at the SoC or subsystem level, this methodology is used to replace the traditional directed tests for better coverage in many testing areas [1]. However, because the system information used in the test environment is not comprehensive enough, many scenarios still need to be tested by the directed tests, for example, the scenarios that require disabling traffic through a specific datapath when testing power/clock gating in different IP combinations or a DUT with missing/broken IP(s). With the growth of the number of IPs integrated in the system [2], the manual setup for these directed testcases is becoming more and more a tedious challenge for the engineers. To help deploying constrained random methodology to these scenarios, a more comprehensive data structure that contains the connectivity info and provides an easy way to access is urged to be implemented in the test environment. This article is proposing a graphical topology info structure, which can be used in the testbench for easily retrieving the system information, and hence, helps the tests automatically setup constraints for stimulus generation to precisely avoid undesired traffic. Optionally, the topology info can be extended to carry extra system info which can grant the tests accessibility for better DV component control.**

I. Introduction

At the SoC or subsystem level, most of tests can apply the constrained random methodology to generate valid end-to-end traffic without paying too much attention to the IPs in the datapath [3]. However, without any information from the connectivity in the entire system, when testing the power gating (PG) or clock gating (CG) features for a certain set of IPs, the verification engineers still need to use a directed testing method and manually setup the stimulus generation on a case by case basis. In order to reach the idle state where power/clock gating can be triggered in the IP(s), the directed tests need to either stop all traffic going into the system or selectively disable traffic going through the IP(s) under test [4]. Nowadays, when the number of IPs integrated in the system has been quickly growing, this kind of directed testing method may introduce more and more coverage holes and become a heavy-duty task to the engineers in their job. Therefore, there is a desire for a test environment that supports CRV in these scenarios in order to improve the verification quality and efficiency.

Furthermore, it is commonly hard to guarantee that all IPs are fully ready at the same time when delivering to the SOC or subsystem. This means some IPs may be missing or broken at the moment when the SoC regression is already up and running. With minimum information for the stimulus generator, the random testing approach will freely generate traffic to all available datapaths, in which some may go through the missing or broken IP. As a result, certain checkers and/or assertions may fire unexpectedly, or even worse, simulation may hang without any verbose debug message. By knowing particular IPs are not ready, the verification engineers still need to spend time on debugging the failing case, disabling checkers or selectively removing some stimulus in the tests with multiple iterations. It is time-consuming with little reward because there are no real bugs. To avoid this situation, if the test environment can leverage information about the missing/broken IPs when generating the stimulus, some pre-emptive actions can be taken to prevent false-positive error reporting in the regression, and hence, will save engineers' time in this task and use it for real problems in the verification.

To solve these problems, a more comprehensive topology info is proposed for the testbench, which will provide system connectivity info to the test environment for better constraining the traffic generation. Optionally, it can be extended to contain extra information about the IPs and the interfaces in the system, so that users can easily access and configure the DV components such as checkers or scoreboards according to the needs. This proposed topology info consists of nodes representing the individual IPs and directional edges representing the connections between two adjacent IPs. In this way, the entire info structure is formed as an edge-node graph in terms of G = (E, V), and hence it is named a graphical topology info structure.

II. The System Info and The Precursor File

The topology info is essentially a database storing the system information. The primary requirement of this info structure is to precisely reflect the connectivity relationship among IPs. Here is an example of the high-level design diagram with connectivity among 12 IPs.

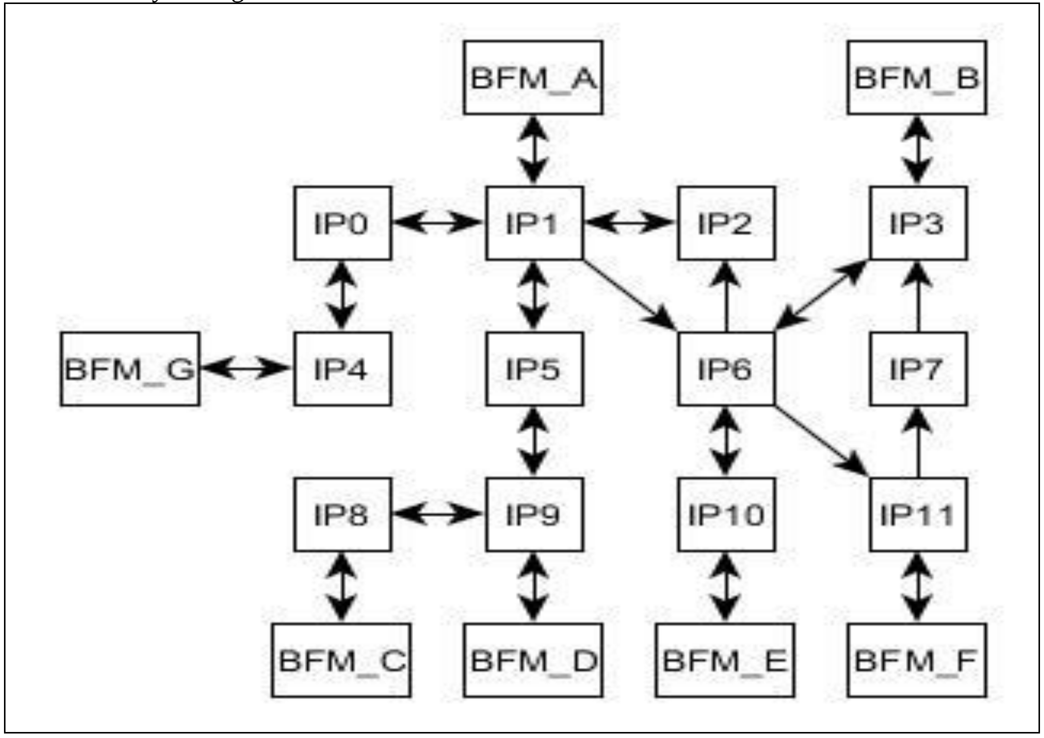

Figure 1. Example of a SoC architecture.

According to the system specification or the high-level design diagram, precursor files can be created to contain information about the IPs and the connections with necessary interface attributes. They can be in any format for convenient use, for example, a plain text file, a json file or the output file from a visual drawing tool such as yEd. Then a user-defined helper script will parse and convert the precursor file(s) into the source codes of the topology info structure. The reason to setup the intermediate files instead of directly creating the source codes of the dv components is to decouple the source of information from its final products. In this way, with different builders in the script, the information in the precursor file(s) can be reused in different languages, different methodologies or different verification platforms. For example, the IP teams are using SystemVerilog/UVM in the test environment while SoC team is using SystemC and the emulation team is using C++. Here is an example of a json file containing IPs and connections information from a system diagram such as Fig. 1.

```
{
   "ip":{
      "IP0": {
         "name": "IP0",
         "is_boundary": 1,
         …
      },
      "IP1": {
         …
      },
      …
   }
   "connection": {
      "ip0_ip1": {
         "name": "ip0_ip1",
         "protocol": "AXI",
         "source": "IP0",
         "destination": "IP1",
         …
      },
      "ip1_bfmA": {
         …
      },
      …
   }
}
```

Figure 2. Example of system info json file.

For the verification flow, either the SoC/subsystem team creates the files and passes to IP teams from top-down, or the IP teams can provide the files from bottom-up when delivering the designs for SoC integration.

III. Graphical Topology Info in UVM

Technically, the topology info structure can be built in various DV methodologies such as C++, SystemC, or UVM. Due to popularity, this section introduces the data structure for the topology info in UVM and shows how the graph is built into the UVM test environment from the precursor file.

A. *The Node: IP Info Class*

An IP info class is the primary element in the graphical structure of the topology info. Each IP info shall accurately represent the corresponding IP. If the IP is at the boundary of the system, which means its inputs and outputs are considered as primary IO and connected to the external components such as a BFM or an iUVC, then it should be marked as a boundary IP. The boundary IP is usually the source or the destination of a datapath, which will be used in the source-destination pair representing a datapath later. To extend the features of the topology info, the IP info class can also contain pointers to the IP's mUVC and the iUVC of its interfaces. Here is an example of the UVM codes for the IP info class:

```
class ip_info extends uvm_object;
   // Name of the IP
   string name;
   // Flag of boundary IP
   bit is_boundary;

   // Pointer to the mUVC
   ip_mUVC muvc;
   …
   // List of pointers to the iUVCs on different interfaces
   ip_iUVC iuvc[$];
   …
```

Figure 3. Example of IP info class.

B. *The Edge: Connection Info Class*

A connection info class is another primary element in the graphical structure of the topology info. It represents the connectivity between two adjacent IPs and includes necessary interface information if needed. Since the connectivity is directional, this class contains the pointers of the source IP and the destination IP. On the other hand, it can also contain the protocol of the interface, the pointer of the iUVC and even the channel of the connection.

```
class connection_info extends uvm_object;
   string name;                 // Name of the connection
   string protocol;             // Protocol of the connection
   string channel;              // Channel of the connection
   ip_info source;              // Pointer of the Source IP
   ip_info destination;         // Pointer of the Destination IP
   connection_iuvc iuvc;        // Pointer of the iUVC of the interface
   …
```

Figure 4. Example of topology info class.

C. *The Graph: Topology Info Class*

The topology info is the container of all IP info classes and connection info classes. In this class, the nodes and the edges have formed a graphical data structure for the design topology. To store the nodes and edges, various lookup tables (hash map) can be constructed to help the lookup in different ways, for example, to look up the IP info by its name or to look up the connection info by the source IP. Some helper functions are also implemented in this class for interfacing with the users.

```
class topology_info extends uvm_object;
   string name;
   string type;

   ip_info ip_name_map[string];
   connection_info_array connection_source_map[ip_info];
   ip_info_array source_destination_map[ip_info][ip_info];
   …

   function void add_ip(ip_info ip);
   function void add_connection(connection_info conn);
   function ip_info get_ip(string ip_name);
   function ip_info_array get_ip_combinations();
   function ip_info_array get_adjacent_ip(ip_info ip);
   function ip_info_array get_dest_from_source(ip_info ip);
   function connection_info_array get_connection(ip_info ip);
   function datapath_info get_datapath(ip_info ip);
   …
```

Figure 5. Example of topology info class.

In the UVM environment, the information from the topology info may be needed by other components in the build phase, so all of the classes of this info structures are based on uvm_object. In this way, the topology info can be created in different scopes or different UVM structures when it is needed.

D. *Path Searching Function*

The main function of this graphical topology info is to provide datapath information for the constraints in the tests. When giving an IP, a searching function should return a list of datapaths including the source and destination. Benefiting from the graphical topology of this data structure, there are various algorithms available for path traversing and searching. Without the consideration in the runtime performance, the simplest and most straightforward method for the datapath search is to exhaustively traverse all sources towards the possible destinations and build a lookup table. The lookup table uses the source and destination as the row and the column respectively. Each entry of the table contains all possible datapaths between the corresponding source and destination. Due to the nature of the graphical topology info, the table contains precisely all the datapath info from the system design under test. By looking up an IP in the table, the search function can retrieve a list of datapaths which this IP is in, so that the test can use this list to help constraining the traffic generation accordingly. And also, it can provide the list of possible destinations by giving a source IP. Here is an example of the lookup table referring to the SoC design in Fig. 1. In each entry, the numbers are the numbers from the IP names, and the order of the numbers represents the order of the IPs in the datapath from the source to the destination.

TABLE I
LOOKUP TABLE OF DATAPATHS FROM SOURCE TO DESTINATION

Dest / Src	IP1	IP3	IP4	IP8	IP9	IP10	IP11
IP1	1	1, 6, 3	1, 0, 4	1, 5, 9, 8	1, 5, 9	1, 6, 10	1, 6, 11
IP3	3, 6, 2, 1	3	3, 6, 2, 1, 0, 4	3, 6, 2, 1, 5, 9, 8	3, 6, 2, 1, 5, 9	3, 6, 10	3, 6, 11
IP4	4, 0, 1	4, 0, 1, 6, 3	4	4, 0, 1, 5, 9, 8	4, 0, 1, 5, 9	4, 0, 1, 2, 6, 10	4, 0, 1, 2, 6, 11
IP8	8, 5, 9, 1	8, 9, 5, 1, 2, 6, 3	8, 9, 5, 1, 0, 4	8	8, 9	8, 9, 5, 1, 2, 6, 10	8, 9, 5, 1, 2, 6, 11
IP9	9, 5, 1	9, 5, 1, 2, 6, 3	9, 5, 1, 0, 4	9, 8	9	9, 5, 1, 6, 10	9, 5, 1, 6, 11
IP10	10, 6, 2, 1	10, 6,3	10, 6, 2, 1, 0, 4	10, 6, 2, 1, 5, 9, 8	10, 6, 2, 1, 5, 9	10	10, 6, 11
IP11	11, 7, 3, 6, 2, 1	11, 7, 3	11, 7, 3, 6, 2, 1, 0, 4	11, 7, 3, 6, 2, 1, 5, 9, 8	11, 7, 3, 6, 2, 1, 5, 9	11, 7, 3, 6, 10	11

When investigating IP5, for example, the function just needs to search the entries containing IP5 in the table. To better demonstrate the searching result, the entries containing IP5 are grayed out in TABLE I. Then the function returns the source-destination pairs corresponding to each entry. Here is the list of the pairs found by the function.

- IP1 → IP8; IP1 → IP9;
- IP3 → IP8; IP3 → IP9;
- IP4 → IP8; IP4 → IP9;
- IP8 → IP1; IP8 → IP3; IP8 → IP4; IP8 → IP10; IP8 → IP11;
- IP9 → IP1; IP9 → IP3; IP9 → IP4; IP9 → IP10; IP9 → IP11;
- IP10 → IP8; IP10 → IP9;
- IP11 → IP8; IP11 → IP9;

To store the datapath list above, a datapath_info class can be created to contain the pointers to the source IP and destination IP. Here is an example of the datapath_info class in UVM.

```
class datapath_info extends uvm_object;
   // Name of the datapath
   string name;

   // Pointer to the source IP
   ip_info source;
   // Pointer to the destination IP
   ip_info destination;
   …

   // Helper Functions if needed
   …
```

Figure 6. Example of datapath info class.

E. Construction and Build

The source codes of the UVM data structure including the IP info classes, connection info classes and topology info classes can be generated from the precursor file(s). Depending on the format of the precursor file(s), a helper script can be created to parse the precursor files and produce the corresponding SystemVerilog files. According to the user's needs, the script may generate partial source codes to plug into the test setup, or the entire topology info class with everything built-in. In the following example shown in Fig. 7, a Perl script with the JSON plug-in library is written to generate the source codes for the UVM environment.

```
use JSON;
…

sub generate_uvm_class {
  my json_data = read_json_file();
  my uvm_file = “example_constructor.sv”;
  open (my $uvm_fh, ‘>’, $uvm_file);
  print $uvm_fh “ip_info _ip_info;\n”;
  print $uvm_fh “connection_info _conn_info;\n”;

  # Generate IP info class
  foreach my $block (sort keys %{json_data}) {
    if ($block == “ip”) {
      foreach my $ip (sort keys %{$json_data->{$block}}) {
        # Create new ip info instance
        print $uvm_fh "_ip_info = new($json_data->{$block}->{$ip}-
>{name});\n";
        print $uvm_fh "_ip_info.is_boundary = $json_data->{$block}->{$ip}-
>{is_boundary};\n";
        # Add into topology info
        print $uvm_fh "topology_info.add_ip(_ip_info);\n";
      }
    }
  }

  # Generate connection info class
  foreach my $block (sort keys %{json_data}) {
    if ($block eq "connection") {
      foreach my $conn (sort keys %{$json_data->{$block}}) {
        # Create new connection info instance
        print $uvm_fh "_conn_info = new($json_data->{$block}->{$conn}-
>{name});\n";
        # Lookup ip in topology info and set the point for src/dest
        print $uvm_fh "_conn_info.source = topology_info.get_ip($json_data-
>{$block}->{$conn}->{source});\n";
        print $uvm_fh "_conn_info.destination =
topology_info.get_ip($json_data->{$block}->{$conn}->{destination});\n";
        # Add into topology info
        print $uvm_fh "topology_info.add_connection(_conn_info);\n";
      }
    }
  }

  close $uvm_fh;
}
```

Figure 7. Example for generation functions from the helper script in Perl.

The function in the script creates instances of IP info and connection info classes according to the Json file. The instances are added into the topology info where they will be stored in different hash maps for later lookup. The IP info needs to be built and added first, so that the connection info can retrieve the pointers to the source IP and destination IP.

The script can be extended to generate the codes in different languages or methodologies for different usage. For example, generate_cpp_class function can be written to generate C++ source codes.

In the test environment, topology info shall be built and registered into uvm_confg, so that it can be retrieved easily in different scopes or sequences in the environment. Here is a code example of the build of an IP info class and connecting it with other IPs.

```
class test_base extends uvm_test;
   topology_info topology_info;
   …
   function void build_phase(uvm_phase phase);
      // Build and register the topology info to config db in build phase
      topology_info = topology_info::type_id::create(“topology_info”);
      uvm_config_db#(topology_info)::set(null,”*”,”topology_info”, 
topology_info);
      …
      // Include the constructor of the graph
      `include “example_constructor.sv”;
   endfunction
   function void connect_phase(uvm_phase phase);
      …
      // Set the pointers of mUVC and iUVC in connect phase
      topology_info.get_ip(“IP0”).mUVC = ip0_muvc;
      topology.info.get_connection(“ip0_ip1_axi”).iUVC.push = axi_iuvc_0;
      …
   endfunction
   …

endclass
```

Figure 8. Example of Building the Topology Info.

IV. USE CASE EXAMPLES

The graphical topology info is implemented in test and used for the followings three test cases.

A. *Random Power/Clock Gating Test*

When testing the power/clock gating feature from an IP, the traffic going through the IP needs to be stopped in order to create an idle state for the gating. In a SoC/Subsystem testbench, generating traffic for a certain datapath usually starts from the source such as an input port connected to a BFM or an iUVC, then targets the destination in terms of a specific address or a range, for example, with an aperture, a node ID or a bus/device/function (BDF) number. When trying to stop the traffic on a given datapath with the IP under test in it, the test needs to be aware of the source-destination pair and avoid generating any transactions from the source targeting the destination. A traditional way to accomplish this objective is to list all the possible combinations and manually set up directed tests in which the traffic for a specific datapath will be avoided and others are kept. However, when the number of IPs grows in the system, to achieve full coverage of all the possible gated IP combinations, the work of setting up the directed testcases for DV engineers will become more and more repetitive and challenging. For example, in a system with four IPs named IP1, IP2, IP3 and IP4. There will be 15 gating scenarios to cover. The breakdown is shown in TABLE II.

TABLE II
IP COMBINATION BREAKDOWN

Scenario	**Number of Possible Combinations**	**List**
Individual IP	4	IP1, IP2, IP3, IP4
2-IP Combo	6	IP1+IP2, IP1+IP3, IP1+IP4, IP2+IP3, IP2+IP4, IP3+IP4
3-IP Combo	4	IP1+IP2+IP3, IP1+IP2+IP4, IP1+IP3+IP4, IP2+IP3+IP4
4-IP Combo	1	IP1+IP2+IP3+IP4

Deriving from the sum of n-combinations for all n in the binomial theorem [5], when giving n IPs, the number of total possible PG/CG cases can be calculated with the following equation:

$$f(n) = 2^n - 1. \tag{1}$$

According to (1), the number of PG/CG testcases increases exponentially with the linear increase of the number of IPs. Referring to the system shown in Fig. 1, when there are 12 IPs in the system, there may be 4095 directed tests need to be set up for full coverage on the combination of gated IPs.

To offload the work to the constrained random tests, the graphical topology info class can be setup in the environment. The topology info can easily search for all the datapaths containing the given IP(s) and provide the source-destination info for the constraints used in the random traffic generation.

First, the PG/CG test creates the gating scenario by randomly picking a combination of IPs to be gated. Then through the topology info look up, it retrieves a list of excluded datapaths that contain the given IP. The excluded datapath is in terms of the source-destination pairs used in the constraints for traffic generation. When generating the stimulus, the traffic on all excluded datapaths will be properly avoided in the randomization.

Here is a UVM example of a random test using the graphical topology info.

1. Use a virtual sequence to host the constraint and spawn sequence with different source.

```
// pg_virtual_sequence.sv
class pg_virtual_sequence extend uvm_sequence;
   ip_info source;
   ip_info_array dest_list;
   ip_info_array excluded_dest;
   …

   // Helper functions
   function void set_source(ip_info ip);
      source = ip;
   endfunction
   function void set_destination(ip_info_array ip_array);
      dest_list = ip_array;
   endfunction
   function void set_excluded_destination(ip_info_array ip_array);
      excluded_dest = ip_array;
      // Remove the destinations from the destination list
      refine_dest();
   endfunction

   // Body
   virtual task body();
      …
      // send_traffic method will randomize the targeting address from
the refined dest list and send transactions from the source IP
      if (dest_list.size() != 0)
         send_traffic(source, dest_list);
   endtask
   …

endclass
```

Figure 9. Example of power gating virtual sequence.

2. Create a test which extends from the test_base where the graphical topology info is built. The test gets random IP combination(s) and lookup topology info for an excluded list of the datapaths.

```
// pg_test.sv
class pg_test extend test_base;
   datapath_info excluded_datapaths[$];
   …

   virtual task build_phase(uvm_phase phase);
      pg_ip_set = topology_info.get_ip_combinations();
      foreach (pg_ip_set[i]) begin
         // Get the excluded datapath from each IP under test.
         excluded_datapaths.push_back(topology_info.get_datapath(pg_ip_set[i]));
      end
      foreach (all_sources[i]) begin
         automatic int var_i = i;
         fork
         begin
            ip_info_array excluded_destination;
            ip_info_array all_dest_list;
            // Get all the possible destinations from the source
            all_dest_list = topology_info.get_dest_from_source(all_sources[var_i]));
            // Get the excluded destinations from the datapaths
            foreach (excluded_datapaths[j]) begin
               // Only get the destination from the same source
               if (excluded_datapaths[j].source == all_sources[var_i])
                  excluded_destination.push_back(excluded_datapaths[j].destination);
            end
            v_sequence.set_source(all_sources[var_i]);
            v_sequence.set_destination(all_dest_list);
            v_sequence.set_excluded_destination(excluded_destination);
            repeat(10) begin //send 10 transactions
               v_sequence.randomize();
               v_sequence.start();
            end
         end
         join_none
      end
      wait fork;
      …
   endtask
   …

endclass
```

Figure 10. Example of a power gating test using graphical topology info.

3. Run the test.

In the example above, the test randomly selects one scenario and creates traffic with ten transactions from each source. With the excluded list containing the IP(s) to be gated in the test, the virtual sequence can remove the IPs from the destination list accordingly, so that the unwanted destination address will be excluded when the traffic is generated from the source. In the test, fork-join is used for spawning traffic from all sources simultaneously. In this way, we achieve the objective of this power/clock gating test, which is to stop traffic through the IP(s) under test and keep other traffic running.

According to the testing requirements, the user can easily expand the test to exercise multiple scenarios at a time and randomize the number of transactions from each source. When applying this setup to test the system shown in Fig. 1, instead of running approximately four thousand directed tests in the regression, only one random test is needed to be run multiple times towards the coverage goal. The test can also be plugged into a current power-aware verification flow, with a coverage-driven methodology for the coverage targets with very little overhead. With the tradeoff of some extra regression runs, it saves tremendous engineering time spent on the manual setup.

In conclusion, with the help of this graphical topology info, one constrained random test can replace hundreds or thousands of directed tests or pre-generated tests, furthermore, it is reusable in different SoCs and scalable with different numbers of IPs in the system.

B. *Missing IP or Broken IP in DUT*

When integrating multiple IPs into an SoC, it is difficult to guarantee that all IPs are ready and work properly at the first place before SoC regression starts. Sending traffic through a missing IP such as an RTL shell or a broken IP may result in some unexpected error reports, or even worse, hanging in the simulation. The engineers need to spend tremendous time and energy to triage and debug the problem, though it is already known that the IP is missing or broken.

After setting up the graphical topology info, the user just needs to provide the list of missing IPs and/or broken IPs when configuring the tests before running, then tests will retrieve the excluded datapath and avoid generating traffic through the given IPs. The list of the not-ready IPs can be generated by marking the IPs in the precursor files, or setting them through the uvm_config_db. The above-mentioned virtual sequence and test can be reused with very minor alternation for these testcases. In this way, it utilizes the known system information for the test to reduce the engineering time in dealing the false positive error reports from the regressions.

C. *Checker Control*

Optionally, besides helping the constrained random test with the connectivity info, the topology info can be expanded to carry more IP and interface information. For example, the pointers to the mUVC of the IP, iUVC of the connection or the control signals of the assertions. When a certain checker or scoreboard needs to be configured or disabled for a specific test, it is easy for the users to retrieve the corresponding DV components from the topology info and apply the changes. With the accessibility through the topology info, the users can minimize the hardcoding or guarding macros for the customized setup in some specific testing scenarios.

V. LIMITATION AND FUTURE WORK

This paper only discusses one of the dataflow scenarios in an SoC which sends end-to-end transactions. Sometimes, SW programming, RAS or interrupts will also trigger transactions through IPs. In this case, the source IP may not be explicitly at the boundary of the system. Potential improvements need to be implemented in the topology info to deal with these cases.

On the other hand, the path searching function assumes that there is only one datapath from the source to the destination. However, some systems may have multiple datapaths between two IPs depending on the targeting address ranges. To handle this, the topology info needs to contain more attributes in the IP and the connection including the routing information. The lookup table in the path searching function also needs be extended to use each range in the IP as the destination. Furthermore, though the case with a loop in the datapath is very rare in most SOCs, the path searching function in graphical topology info can also be improved to handle this corner case when it is needed.

VI. SUMMARY

By implementing the graphical topology info containing all the necessary system information, the SoC/subsystem tests can now easily leverage the constrained random approach to generate traffic to the desired datapath. In this way, engineering effort can be saved from the manual work in creating directly tests or configuring the verification environment for some special needs. The current applications are for power/clock gating tests and DUT with missing/broken IPs. However, this graphical topology info can be easily extended to embrace other functionality to help testing in more areas such as QoS or performance.

REFERENCES

[1] J. Chen, "Applying Constrained-Random Verification to Microprocessors", EETimes, SoC Designline, Dec 10th 2007

[2] H. Foster, "The 2016 Wilson Research Group Function Verification Study", Mentor Graphics Verification Horizons BLOG, Jan 3rd 2017

[3] E. Worthman, "It's All IP In An SoC", Semiconductor Engineering, IoT, Security & Automation, June 5th 2014

[4] M. Keating, D. Flynn, R. Aitken, A. Gibbons, K. Shi, "Architectural Issues for Power Gating", The Engineer's Portal to Green Design, vol 100909, originated from Chapter 6 in Low Power methodology Manual for System-on-Chip Design (New York: Springer 2007)

[5] R. Graham, D. Knuth, O. Patashnik, "Concrete Mathematics: A Foundation for Computer Science", 2nd ed., Addison-Wesley, 1994