IP-XACT Based PSS Modeling For Shift-Left SoC Verification

Moonki Jang , Yonghyun Yang , Kangho Lee , Yejin Lee , Youngchan Lee, Sunil Roe , Youngsik Kim , Seonil Brian Choi
Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18488, Korea, moonki.jang@samsung.com , yh1597.yang@samsung.com , kangho71.lee@samsung.com, yejin6.lee@samsung.com, yc0325.lee@samsung.com, sunil.roe@samsung.com, ys31.kim@samsung.com, seonilb.choi@samsung.com

***Abstract*- Modern SoC designs frequently exhibit functional corner-case bugs that are difficult to uncover in pre-silicon environments. This work introduces an automated shift-left verification methodology that generates a Portable Stimulus Standard (PSS)[1] model directly from IP-XACT[2] data. By extracting hierarchy, interfaces, signal behaviors, and end-to-end connectivity, the method builds a unified PSS modeling database and automatically generates PSS components and actions without manual realization-layer work. The resulting model enables systematic behavior exploration and significantly expands functional coverage, achieving approximately a 30% increase in total functional coverage bins through automatic enumeration of design-derived functional conditions. Furthermore, the case study demonstrates how the proposed framework enables the discovery of corner-case conditions in the pre-silicon environment, and illustrates why PSS represents the most suitable verification methodology in the emerging era of AI-driven test generation.**

I. INTRODUCTION

Semiconductor design verification has long sought to identify complex corner-case bugs at the pre-silicon stage, rather than discovering them during silicon validation. Early detection significantly reduces the cost of bug fixes, a concept widely known as the shift-left strategy

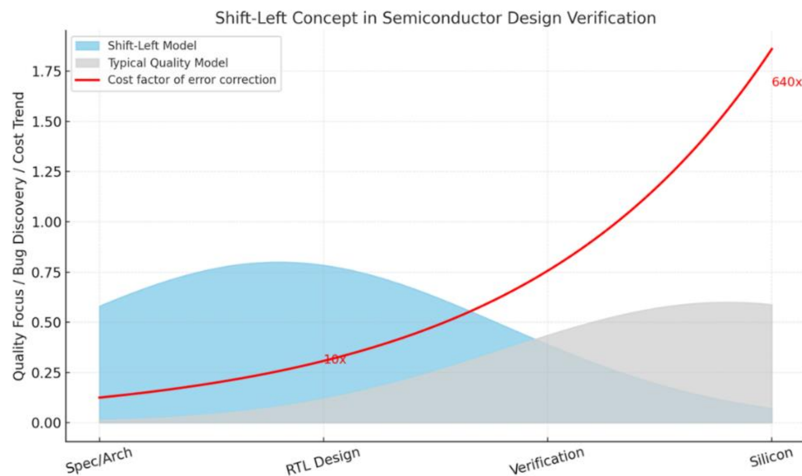

Figure 1. Illustrates the shift-left requirement in semiconductor design verification

Historically, shift-left strategies have emphasized accelerating simulation to reproduce bug-triggering conditions observable at the silicon level. Despite advances by EDA vendors, reproducing extremely rare corner-case conditions that occur only once in several days remains highly constrained in pre-silicon environments.

Two major challenges prevent comprehensive pre-silicon corner-case verification.

A. Specification incompleteness:

Human-written design specifications are inherently incomplete. Functional coverage bins are derived from specifications, which typically describe high-level functions but omit many hardware conditions occurring at the SoC level. As a result, randomization has traditionally been used to explore corner cases. However, experience has shown

that real silicon bugs rarely emerge from simple randomized sequences. Coverage should not only measure verification progress but also identify missing verification scenarios. This requires moving beyond specification-based coverage to a methodology that enumerates all functional features and hardware conditions.

B. Verification environment limitations:

Pre-silicon verification is significantly slower, operating at roughly three orders of magnitude below silicon performance, even when simulation accelerators are employed. As a result, the number of executable scenarios is limited, and it is nearly impossible for engineers to manually construct all hardware conditions required for thorough verification. Nevertheless, unlike silicon, the pre-silicon environment provides unique advantages, such as fine-grained signal-level control, comprehensive observability, and the ability to execute hundreds or even thousands of tests in parallel. These features enable more flexible test strategies, motivating an approach in which concise tests are deliberately designed to construct the desired hardware conditions.

To address these two challenges, we investigate the use of IP-XACT design descriptions for PSS modeling. IP-XACT is a standardized technical format defined by IEEE 1685 that describes design components using a standardized metadata format, improving IP reusability and enabling automation of RTL integration. By directly importing design integration information from IP-XACT into PSS, the traditionally complex and labor-intensive process of manually specifying hardware details is significantly simplified and automated. Furthermore, functional coverage is substantially expanded through IP-XACT port behavior analysis. In this paper, we demonstrate how SoC internal structures can be systematically incorporated into PSS models using IP-XACT information, thereby enabling automated model and test generation, and show how this approach achieves a significant expansion of functional coverage.

II. IP-XACT Based PSS Modeling Strategy

The Portable Stimulus Standard (PSS) has primarily focused on modeling functional behaviors and flows to describe test intent. Consequently, SoC-level structural information such as hierarchy or control/data paths has been underutilized.

For this reason, the DVCon US 2018 paper *“Building Portable Stimulus Into Your IP-XACT Flow”*[3] investigated an approach in which pre-defined PSS actions were embedded into IP-XACT, enabling PSS modeling that directly reflects the IP-XACT component tree. In this work, we take a step further by analyzing design specifications, including IP-XACT, to identify all potential functional cases that may occur at the SoC level. Based on this analysis, we present our results on automatically generating complete PSS models, including components and actions, capable of verifying these functional cases.

Figure 2 provides an overview of the proposed IP-XACT based PSS modeling strategy. Information required for PSS modeling is extracted from multiple IP-XACT XML files and consolidated into a unified, JSON-based PSS modeling database. This database captures the SoC hierarchy, inter-component connectivity, port-level behaviors, and functional relationships derived directly from design descriptions. Based on this database, a structural PSS model is automatically generated, representing the SoC topology and its functional interaction space at an abstract level. In parallel, an action library, which is independently authored and organized according to IP-level functionality, is analyzed and selectively combined with the generated structural model. Through this integration step, instance-level actions are associated with the corresponding port behaviors and connectivity information stored in the modeling database, resulting in a complete project-specific PSS model. At the realization layer, which generates executable tests, the PSS modeling database is further referenced to incorporate RTL paths, IP details, and connection maps from the actual design. This approach enables automatic binding between abstract actions and design-specific realization information, eliminating the need for engineers to manually retrieve and encode RTL paths or integration details from design specifications. As a result, both structural information and reusable action semantics are systematically incorporated into the final PSS model in a fully automated manner.

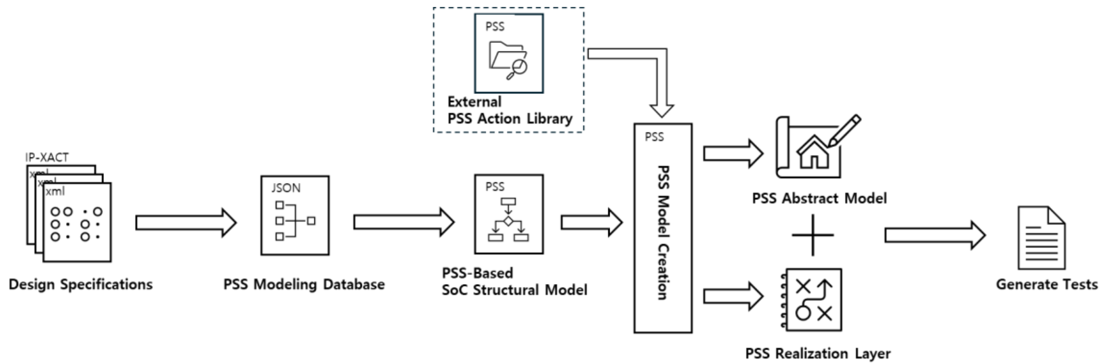

Figure 2. Concept of IP-XACT based PSS modeling

We utilized two scripts in this workflow: an interpretation script that extracts the information required for PSS model construction from IP-XACT and builds the PSS modeling database, and a generation script that produces the PSS model from this database. Our objective was to establish an environment in which PSS models could be automatically derived from IP-XACT without any manual effort. To avoid relying on unnecessarily complex scripts, we carefully designed both the database schema and the overall modeling architecture. In the following section, we describe this process in detail and present the resulting artifacts.

A. SoC Connection Map Modeling

At the SoC level, all functional behaviors originate from the interconnections among internal IPs and blocks. Therefore, to verify various functional conditions that may occur at the SoC level, it is crucial to construct a PSS model that reflects the hierarchical structure and interconnections among IPs and blocks within the SoC. In this paper, a simplified example SoC is presented to illustrate how PSS modeling can be achieved using IP-XACT information.

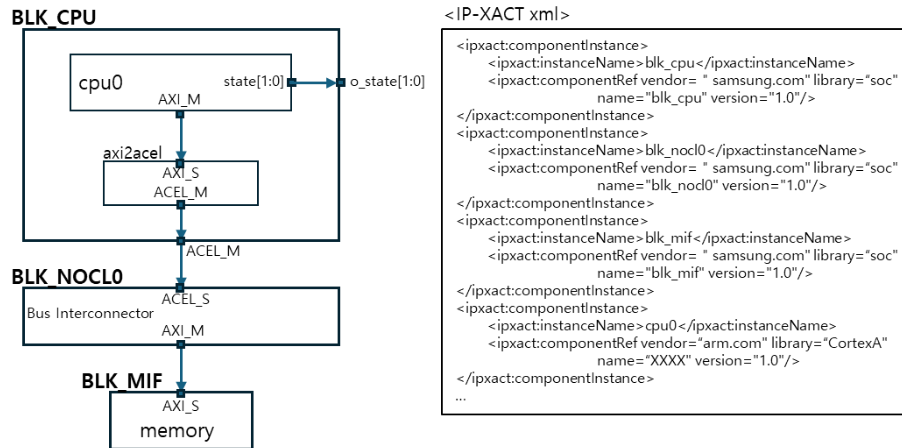

Figure 3. Block diagram and IP-XACT of Example SoC

We first analyzed the instance hierarchy defined in IP-XACT and incorporated the SoC design hierarchy, including each block and its internal IP composition, into the PSS modeling database. Figure 4 presents a simplified structural view of the JSON-formatted PSS modeling database derived from the IP-XACT description of the example SoC. The complete JSON listing is provided in Appendix A. As shown, hierarchical instance keys are used to represent the structural relationships among SoC instances. Each block and its internal instances were assigned unique identifiers to enable easy access to instance attributes within the PSS environment.

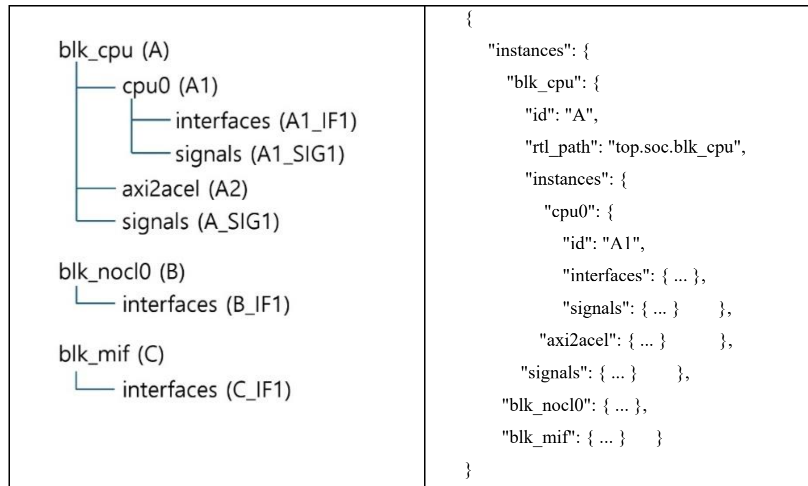

Figure 4. PSS modeling database(JSON)

In addition to the SoC hierarchy, the PSS modeling database also captures the connectivity information among instances. By aggregating connection data from the IP-XACT descriptions, we generated a connections entry, as shown in Figure 5. Each connection specifies the source and destination ports, the connection type, and a unique identifier that enables direct utilization within the PSS model. Furthermore, by consolidating these connections, we constructed a Path Catalog that records the complete traversal path from each source port to its corresponding destination port. Together, these elements form the Connection Map that fully represents the inter-instance connectivity of the SoC.

The connection map not only illustrates simple source-to-destination relationships but also provides the PSS model with hierarchical information of IPs, blocks, and the overall SoC. Based on this information, PSS can generate test scenarios that systematically account for the complete SoC structure.

Generated Connections	Generated Path Catalog
"connections": [	"path_catalog": [
{	{
"id": "CONN_01",	"path_id": "PATH_01",
"kind": "interface",	"src": { "inst_id": "A1", "if_id": "A1_IF1" },
"protocol": "AXI4",	"dst": { "inst_id": "C0", "if_id": "C0_IF1" },
"src": { "inst_id": "A1", "if_id": "A1_IF1" },	"route": [
"dst": { "inst_id": "A2", "if_id": "A2_IF1" } },	{ "hop_inst": "A2", "via_if": ["A2_IF1", "A2_IF2"] },
{	{ "hop_inst": "B0", "via_if": ["B0_IF1", "B0_IF2"] }],
"id": "CONN_02",	"protocol_chain": ["AXI4", "ACE-Lite", "AXI4"],
… },	"rtl_hierarchy": [
{	"top.soc.blk_cpu.cpu0",
"id": "CONN_03",	"top.soc.blk_cpu.axi2acel",
… },	"top.soc.blk_nocl0",
{	"top.soc.blk_mif"] }
"id": "CONN_04",	]
… }],	

Figure 5. Generated Connections and Path Catalog (JSON)

To generate the PSS model from the JSON database constructed from IP-XACT, we developed a package named db_types, as illustrated in Figure 6. This package formalizes the attribute fields of each instance and their interconnections extracted from the JSON database, enabling systematic representation and utilization within the PSS environment.

```
package db_types {
  …
   // Address range used in path permissions
  struct addr_rng_t { ... };   // (lo, hi) allowed address interval

// Instance metadata: name, type, RTL path, hierarchy (parent, depth)
```

```
  struct inst_rec_t { ... };

// Interface/port metadata: protocol, master/slave role, RTL path
  struct if_rec_t { ... };

// Encoded signal value → semantic label (e.g., 0: idle, 1: read)
  struct sig_behavior_t { ... };

// Signal information: direction, width, RTL path, behavior labels
  struct sig_rec_t { ... };

// Internal signal connection: src → dst mapping inside a block
  struct sig_link_t { ... };

// One hop in an end-to-end path: instance + via interfaces
  struct hop_rec_t { ... };

// End-to-end communication path: src/dst interfaces, route, protocol chain, RTL hierarchy, address constraints
  struct path_rec_t { ... };
};
```

Figure 6. Database structure package (db_types.pss)

Figures 7 and 8 show a simplified version of the blk_cpu_db_c component used to construct the database for the blk_cpu block. The full implementation is included in Appendix B. This component consists of a build_db action, which generates the block-level database, and a set of lookup actions used to retrieve specific information from the constructed database. The build_db action populates the database using the structures defined in the db_types package based on the IP-XACT descriptions. Through this mechanism, PSS can automatically generate tests by referencing the structural and connectivity information of each IP instance.

```
import db_types::*;

package blk_cpu_db {
   component blk_cpu_db_c {
      const block_id_t BLK = "blk_cpu";

   // Inventory Size
   const int N_INSTS = 3;  const int N_IFS   = 3;  const int N_SIGS  = 2;  const int N_LINKS = 1;

   // Inventory Table
   array<inst_rec_t, N_INSTS> insts;  array<if_rec_t,   N_IFS>   ifs;  array<sig_rec_t,  N_SIGS>  sigs;
   array<sig_link_t, N_LINKS> sig_links;

   // Data Population
   action build_db {
     exec body {
       // ---- instances ----
       // insts[0]: blk_cpu (BLOCK) at top.soc.blk_cpu
       // insts[1]: cpu0 (RISC_CORE) under blk_cpu
       // insts[2]: axi2acel (AXI2ACE_BRIDGE) under blk_cpu
       // ---- interfaces ----
       // ifs[0]: cpu0.axi_m   — AXI4 master
       // ifs[1]: axi2acel.axi_s — AXI4 slave
       // ifs[2]: axi2acel.acel_m — ACE-Lite master
       // ---- signals ----
       // sigs[0]: cpu0.state[1:0] (out, width=2) with behaviors   0: idle, 1: write, 2: read
       // sigs[1]: blk_cpu.o_state[1:0] (out, width=2), no labels
       // ---- local signal link ----
       // sig_links[0]: cpu0.state[1:0] -> blk_cpu.o_state[1:0]      }    }  }
};
```

Figure 7. Database component of blk_cpu

Furthermore, each database component provides lookup actions, as shown in Figure 8, enabling queries on the instance information stored within the generated database. In addition to blk_cpu, the same methodology is applied to other blocks such as blk_nocl0 and blk_mif, for which corresponding packages (blk_nocl0_db and blk_mif_db) are constructed in an identical manner.

```
…
// Lookup Utilities – The full implementation is included in Appendix B
// list_insts: return the full table of all instances
action list_insts { ... }

// list_ifs: return the full table of all interfaces
action list_ifs { ... }

// list_sigs: return the full table of all signals
```

```
action list_sigs { ... }

// inst_by_id: lookup a single instance using its inst_id_t identifier
action inst_by_id { ... }

// ifs_by_inst: return all interfaces belonging to a specific instance
action ifs_by_inst { ... }

// sigs_by_inst: return all signals belonging to a specific instance
action sigs_by_inst { ... }

// local_sig_links_via: return all local signal links involving a given signal
action local_sig_links_via { ... }
…
```

Figure 8. Database lookup actions of blk_cpu

In addition, we developed a top_db package that includes the path_catalog_c component, as shown in Figure 9. The top_db package constructs all possible paths from each master interface to every reachable slave interface within the build_db action. The path_catalog_c component also provides a set of path lookup actions that allow users to query paths that traverse a specific instance or determine the access route between a given source and destination IP. By leveraging this information, the model can easily generate test scenarios for corner cases, such as situations in which functional operations across multiple IP instances share common traversal paths.

```
// Below presents a condensed view of the path_catalog_c component, highlighting only the structural organization of the database.
// The full implementation is included in Appendix B.
import db_types;

package top_db {
  component path_catalog_c {
    const int N_PATHS = 1;
    path_rec_t paths[N_PATHS];   // End-to-end path records for the SoC

    // build_db: construct the path catalog
    //  - Example: PATH_01 from cpu0 AXI4 master → AXI2ACE bridge → NoC → MIF
    //  - Fills path_id, src_if/dst_if, route, protocol_chain, rtl_hierarchy, allow ranges
    action build_db { ... }

    // paths_via_if: return all paths that traverse a specific interface
    //  - Match if the interface appears as src_if, dst_if, or any via_if in the route
    action paths_via_if { ... }

    // paths_src_to_dst: return direct paths from a given src_if to a given dst_if
    //  - Filter catalog entries where src_if == src_if and dst_if == dst_if
    action paths_src_to_dst { ... }  }   }
endpackage
```

Figure 9. Contents of top_db package

The connection map constructed in this manner is incorporated into the actual PSS test model, as illustrated in Figure 10. First, the block-specific packages generated earlier are imported, and the databases are instantiated and built in the exec init_up phase. The entry action then utilizes the constructed database information to model the test scenarios. The details of this process are described in the subsequent section on Functional Behavior Modeling.

```
// test_top.pss
import db_types;
import blk_cpu_db;
import blk_nocl0_db;
import blk_mif_db;
import top_db;

component test_top {
  // DB handle
  blk_cpu_db::blk_cpu_db_c       CPU_DB;
  blk_nocl0_db::blk_nocl0_db_c   NOC_DB;
  blk_mif_db::blk_mif_db_c       MIF_DB;
  top_db::path_catalog_c         TOP;

  // Component Initialization: Database Loading
  exec init_up {
    CPU_DB.build_db;
    NOC_DB.build_db;
    MIF_DB.build_db;
    TOP.build_db;
  }
  // Test Entry Point
  action entry {
     activity {
```

```
    …
  } }
}
```
Figure 10. Generated test_top component

C. Functional Behavior Modeling

Through the SoC Connection Map modeling described above, we were able to construct a database that captures the structural hierarchy and interconnections among the blocks and their corresponding IP instances within the SoC. Based on this information, we developed a method to systematically identify all functional behavior conditions that can arise within the SoC. This approach overcomes the limitations of traditional verification, which typically focuses on validating individual functional operations, and enables the generation of test scenarios that account for complex combinations of functional interactions across the SoC.

1) Port Scan

All functional conditions that may arise within an SoC originate from the numerous connectivity relationships among signal ports. Through the Connection Map modeling process described earlier, we constructed a database of all instance ports extracted from IP-XACT. The port-scan process begins by enumerating the connectivity between block instances, where each connection is represented as a pair consisting of a source port and a destination port. The source port initiates an operation that influences its counterpart, while the destination port receives functional effects from external inputs.

These connections may appear as ad-hoc signal interactions such as the o_state[1:0] example, where a toggling signal communicates with another block, or as standardized interface-based interactions such as AXI, ACE, or CHI, in which multiple signals operate according to a defined protocol. We categorized both types and encoded them into the database. Using this database, the list of ports involved in the connectivity between any two blocks can be easily obtained.

2) Behavior Search

Once the list of connected ports is obtained through the Port Scan process, the next step is to perform a Behavior Search. Behavior Search enumerates all possible operations that can occur at each port, enabling the prediction of functional conditions that may arise as a result of these port behaviors. The procedure differs for ad-hoc ports and interface ports. For ad-hoc ports, where no predefined communication rules exist, the Behavior Search is performed by examining the port descriptions in the design specification and generating behavior keys accordingly, as illustrated in the following figure.

In contrast, interface ports that communicate according to predefined protocols are treated as an interface group, and the transaction types defined by the corresponding protocol are incorporated into the model as the behavior list.

```
"signals": {
    "state": {
      "id": "A1_SIG1",
      "type": "ad-hoc",
      "name": "state",
      "dir": "out",
      "width": 2,
      "rtl_path": "top.soc.blk_cpu.cpu0.state",
      "behaviors": {
        "0": { "name": "idle" },
        "1": { "name": "write" },
        "2": { "name": "read" }
      }
    }
},
```

```
"interfaces": {
    "axi_m": {
      …
      "behaviors": {
        "AW": {
          "name": "axi_write_address",
          "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.AW"         },
        "W": {
          "name": "axi_write_data",
          "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.W"         },
        "AR": {
          "name": "axi_read_address",
          "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.AR"         },
        "R": {
          "name": "axi_read_data",
          "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.R"         },
        "B": {
          "name": "axi_write_response",
          "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.B"         }    }
    }    }
```
Figure 11. Behavior modification for Ad-hoc and Interface Port

Because a destination port operates in a passive role by receiving inputs, we do not explicitly maintain port-behavior lists for destination ports. However, when a behavior action occurs at a source port, the possible states that the destination port's block may enter are captured as part of the behavior search results and reflected in the model to enable the construction of more diverse functional conditions. These states include the idle state, the busy state in which other functions are active, and clock- or power-gated states. In addition, if the block supports error or recovery mechanisms, such state information is also incorporated through the behavior search process.

```
// The full implementation is included in Appendix B.
"dst_states": [
    {
      "name": "idle",
      "code": 0,
      "description": "No outstanding transactions; ready to accept a new AXI write."
    },
    …
]
```

Figure 12. Example of destination state

3) Action Matching

The final step of the Functional Behavior Modeling process is to associate the behaviors identified from the connectivity analysis with the corresponding PSS actions.

In PSS, an action represents the smallest unit of functionality, and test scenarios are constructed by composing these actions. Each PSS action is modeled with an abstract layer that expresses its functional intent and a realization layer that defines its implementation for the target platform. In existing flows, even when the same IP functionality is reused, differences in SoC integration across products often require modifications to the realization layer. As a result, actions implementing identical functionality must be recreated for each project, leading to significant inefficiency.

By modeling PSS actions such that their realization layer references the PSS modeling database constructed from IP-XACT, we can significantly reduce project-specific dependencies. With this approach, actions are maintained solely at the abstract layer to express their functional intent, while the realization layer is implemented through an action library that consults the IP-XACT–derived database. Consequently, new actions no longer need to be created for each project, and the previously defined action library can be reused directly during test generation for new designs.

Figure 13 illustrates how the action library is managed and utilized within the proposed framework. The action library mirrors the hierarchical structure of IP-XACT instance information, allowing engineers to easily locate the complete set of actions associated with a specific instance, as shown in the figure. Rather than being used directly, the action library is referenced to extract only the required information into a project-specific model. This approach enables the action library to be maintained independently of individual projects or test environments, thereby avoiding tight coupling to specific verification setups.

In the example shown in Figure 13, the CPU action library arm_cortexa_xxxx_1.0.pss is extended by incorporating an action database, from which a project-level model (cpu.pss) is generated. In addition, the action library metadata is propagated into prj_cfg.yaml and action_registry.yaml, enabling systematic test generation and facilitating structured review of action usage and coverage.

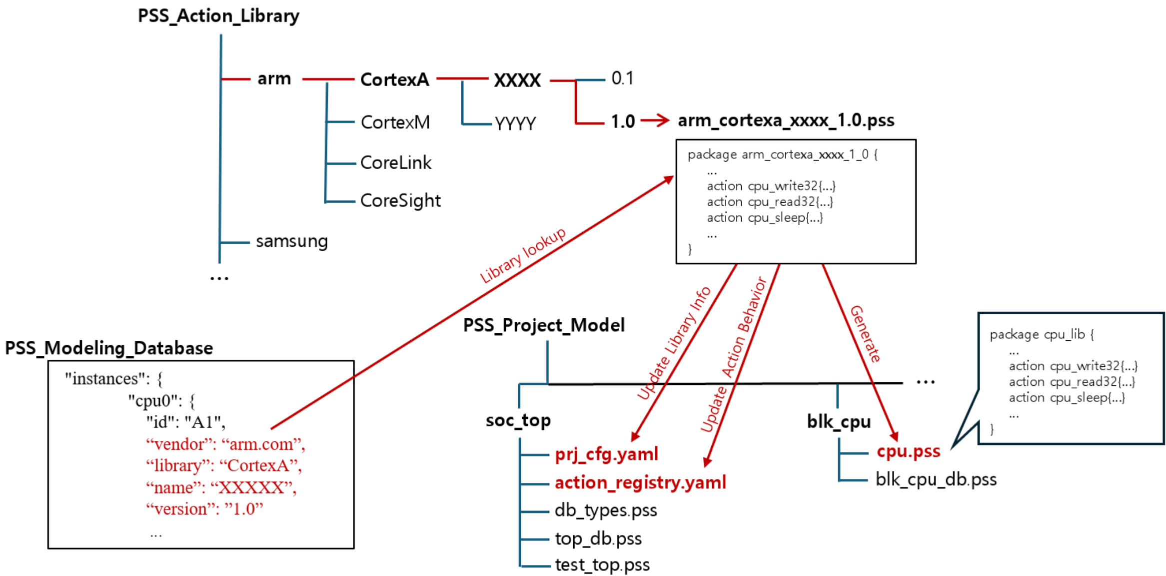

Figure 13. Usage flow of action library

Figure 14 illustrates how actions in the action library are associated with the behavior entries defined in the JSON database. When authoring the action library, we adopted a methodology in which the port information and corresponding behavior descriptions of a given instance are embedded in the action definition as commented YAML-formatted annotations, as shown in the figure.

During the generation of an action library for an individual instance, the IP-XACT based port scanning and behavior search processes are identical to those used in constructing the PSS modeling database. However, when generating actions that describe functional operations, an additional step is performed in which a design specification written in a predefined and structured format is parsed. In this step, the port and function descriptions are interpreted to construct the structural skeleton of each action according to a fixed template.

The commented YAML-formatted metadata associated with each action is generated during the instance-level modeling process and is used to relate the action definitions to the corresponding port behaviors stored in the JSON database.

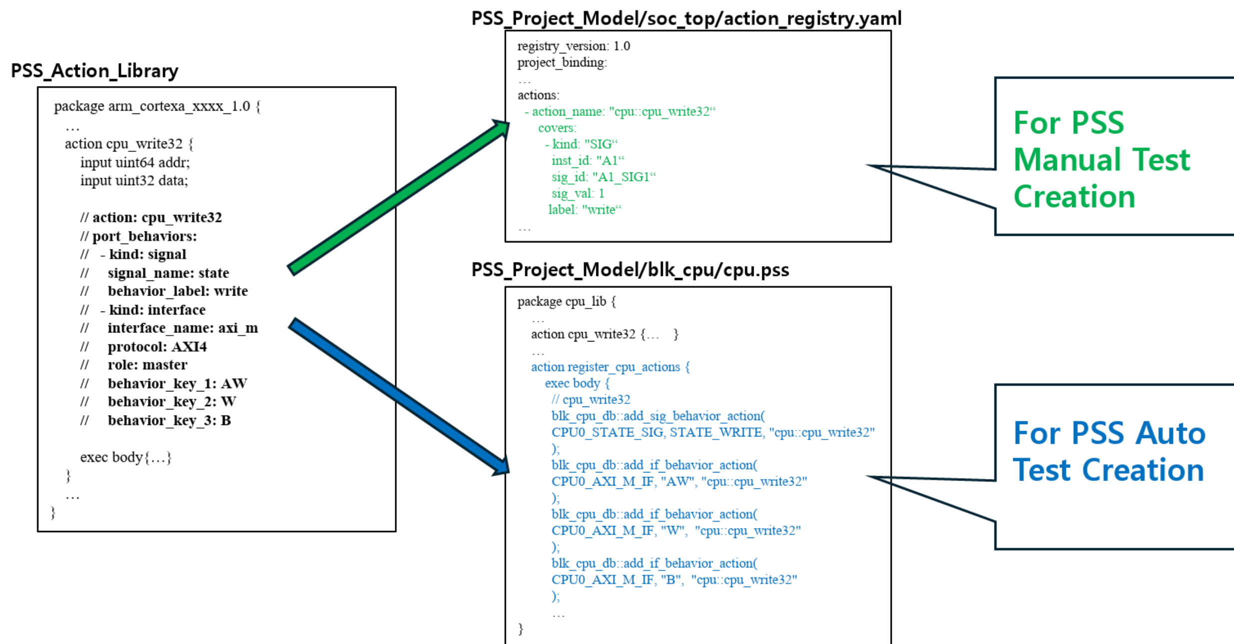

Figure 14. Example of action matching flow

In the workflow illustrated in the figure, the process of generating the project model cpu.pss from the action library includes an explicit step that associates instance-level actions with the SoC-level behavior list. Through this association, the model captures the set of actions capable of triggering specific port behaviors. As a result, all test cases that include a given port behavior can be generated systematically by specifying only behavioral constraints, without manually enumerating individual test sequences.

The action matching for port behaviors also produces an auxiliary database in YAML format, referred to as action_registry, highlighted in green in the figure. This database is not a PSS model itself, but rather an externalized representation of the behavior matching table that is internally maintained within the PSS model structure. By exposing this information outside the PSS model, the action_registry enables structured review of the generated model and facilitates the use of external scripts to construct alternative forms of test scenarios beyond those directly generated by PSS.

The details of our experimental results based on this approach will be presented in the subsequent case study section.

III. Case Study on Practical Applications of the Portable Stimulus Standard (PSS)

Using IP-XACT information to construct a PSS model enables the generation of test scenarios by leveraging the lookup actions shown in Figure 8 and Figure 9. This capability provides an efficient mechanism for identifying signal-level functional conditions required for a given test. In other words, when a specific PSS action is executed, the PSS model supplies a database that describes which port behaviors are triggered and how those behaviors propagate to or affect other instances across different blocks.

To demonstrate how such an IP-XACT-based PSS model can be utilized in practice, we apply it to our previously published PSS methodology presented at DVCon and explain its integration within that framework.

A. Advanced sequence modeling techniques using IP-XACT

Through the Functional Behavior Modeling described above, we derived a comprehensive list of functional behaviors exercised at the SoC level. In particular, by incorporating the possible state information of destination instances into the behavior model, we were able to enforce more precise and expressive functional conditions. This process enabled us to expand functional coverage by identifying and adding functional conditions that were not explicitly specified in the design specification but were revealed through systematic port scanning and behavior-based search.

However, most of the corner-case bugs we have encountered exhibit highly complex and rarely reproducible sequence conditions. Representative examples include deadlocks triggered by a snoop request arriving precisely when a bus-fabric write-buffer overflow induces a stall in the write channel; state-machine malfunctions caused by wake-up interrupts asserted during power-down configuration; and race conditions arising from simultaneous write transactions issued by multiple coherent masters. Such scenarios are notoriously difficult to detect in pre-silicon environments. A common characteristic of these corner cases is that they are predominantly induced by subtle timing conditions at the signal level, which are activated only when specific functional operations are executed. For such corner cases, the typical approach at the silicon level is essentially to rely on chance: engineers exercise various related functionalities over an extended period, hoping that the triggering conditions will eventually manifest.

Reproducing such conditions in a pre-silicon environment remains difficult. Nevertheless, relying on long-duration random testing, as is often done at the silicon level, is neither effective nor appropriate because the pre-silicon environment provides full visibility and controllability at the signal level.

Motivated by these challenges, we previously presented a methodology for reproducing corner-case conditions using PSS at DVCon US 2022, titled "PSS Action Sequence Modeling Using Machine Learning."[4] In the present work, we extend that methodology by leveraging an IP-XACT-based PSS model, which enables a more advanced generation of signal-access sequences directly derived from IP-XACT design information.

The following figure summarizes the work previously presented at DVCon US 2022.

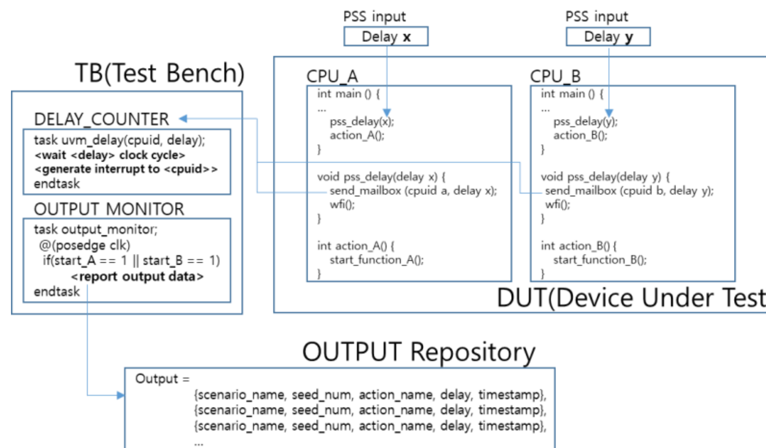

Figure 15. Structure diagram of PSS action sequence model

What we aimed to demonstrate in this example was a PSS sequence model capable of generating tests in which the transactions triggered by action_A and action_B occur precisely at the same time. To realize this behavior, we previously implemented a dedicated delay counter for manually aligning the sequence timing, along with a signal monitor designed to verify whether both transactions were initiated simultaneously, based on the design specification.

In the present work, by incorporating the design specification represented in IP-XACT directly into the PSS model, we were able to implement the port-behavior monitors associated with action execution in a significantly more automated and efficient manner.

```
// sig_monitor.pss

// start_monitor: resolve target RTL signal and invoke SV monitor
action start_monitor {
  activity { resolve_from_db; }

  exec body {
    // Start monitoring "mon_rtl_path" for the given behavior value/label
```

```
    start_sig_behavior_monitor(mon_rtl_path,  mon_behavior_val, mon_behavior_label);
  }
}
…
// sig_behavior_monitor.sv

// Encoded CPU state values (idle / write / read)
typedef enum logic [1:0] {
  CPU_STATE_IDLE  = 2'b00,
  CPU_STATE_WRITE = 2'b01,
  CPU_STATE_READ  = 2'b10
} cpu_state_e;

// Interface + path resolver (details omitted)
// interface state_mon_if (input logic clk); logic [1:0] state; endinterface
// extern function state_mon_if get_state_if_from_path(string rtl_path);

// Start a background monitor for the given RTL path and behavior
task automatic start_sig_behavior_monitor(string rtl_path, int    behavior_val, string behavior_label);
  state_mon_if sif     = get_state_if_from_path(rtl_path);
  cpu_state_e  expected = cpu_state_e'(behavior_val[1:0]);

  fork
    forever begin
      @(posedge sif.clk);
      if (sif.state === expected)
        $display("[MON] %0t : Observed %s (%0d) on %s",
                 $time, behavior_label, behavior_val, rtl_path);
    end
  join_none
endtask
```

Figure 16. Example of signal monitor

Furthermore, by leveraging the information contained in the PSS modeling database, it becomes possible, even though this capability is not defined in the PSS standard, to generate assertion checkers for PSS action behaviors through an external scripting flow and apply them within the testbench, as illustrated below.

```
module action_behavior_assertion #(
  parameter int EXPECTED_STATE = 2'b01,  // expected CPU state
  parameter int MAX_LATENCY    = 8       // allowed cycles to reach it
) (
  state_mon_if mon_if
);
  typedef enum logic [1:0] {
    CPU_STATE_IDLE  = 2'b00,
    CPU_STATE_WRITE = 2'b01,
    CPU_STATE_READ  = 2'b10
  } cpu_state_e;
  localparam cpu_state_e EXPECTED = cpu_state_e'(EXPECTED_STATE[1:0]);
  // Property: when action_trig rises, state must reach EXPECTED within MAX_LATENCY cycles
  property p_action_behavior;
    @(posedge mon_if.clk) disable iff (!mon_if.rst_n)
      mon_if.action_trig |-> ##[0:MAX_LATENCY] (mon_if.state == EXPECTED);
  endproperty
  // Assertion
  a_action_behavior: assert property (p_action_behavior)
    else $error("[ASSERT] Action-behavior violation: state=%0b expected=%0b",
                mon_if.state, EXPECTED);
endmodule
…
// Example instantiation
state_mon_if cpu0_state_if(.clk(clk), .rst_n(rst_n));
action_behavior_assertion #(
  .EXPECTED_STATE(2'b01),  // write
  .MAX_LATENCY   (8)
```

```
) u_cpu0_write_assert (
  .mon_if(cpu0_state_if)
);
```

Figure 17. Example of generated assertion from PSS action behavior

By leveraging IP-XACT information, we enhanced our previous action-sequence model, enabling us to efficiently reproduce and verify corner-case conditions that were previously difficult to trigger in a pre-silicon environment. The table below presents the difference in the number of functional coverage bins identified for blk_cpu, where the host CPU resides, using the new PSS model. Overall, the total number of functional coverage bins increased by approximately 30%, and, notably, the number of cross-function bins increased by an impressive 434% due to the adoption of the advanced sequence model described earlier.

TABLE I
NUMBER OF FUNCTIONAL COVERAGE BINS

Functional Cover group	Previous #of function bins	Latest #of function bins	Difference (%()
Accessibility	224	249	+11.1%
Block functions	126	131	+3.9%
Bus interface	80	107	+33.7%
Clock	147	183	+24.4%
Power	104	143	+37.5%
Reset	19	22	+15.7%
Interrupt	295	345	+16.9%
Security	160	178	+11.2%
Cross Functions	38	203	+434.2%
Total:	**1193**	**1558**	**+30.5%**

B. AI assisted SoC Risk Prediction techniques

The work presented at DVCon US 2023, titled "Early Detection of Functional Corner-Case Bugs using Methodologies of ISO 26262"[5], proposed a methodology for predicting design vulnerabilities and establishing verification plans by deriving a Systematic Failure Severity Level (SFSL). This approach was inspired by the risk assessment process defined in ISO 26262[6], where the Automotive Safety Integrity Level (ASIL) is determined based on severity-related criteria.

As illustrated in the figure below, the risk prediction process evaluates various risk factors to estimate which IP blocks and corresponding functionalities are most likely to exhibit potential vulnerabilities.

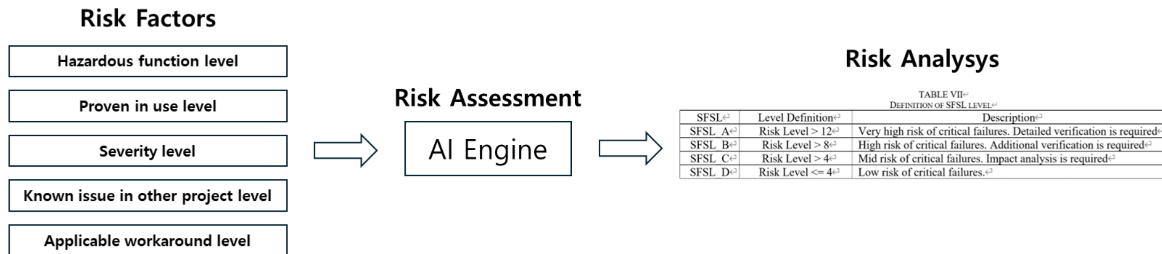

Figure 18. Risk prediction flow

To enable early identification of functional vulnerabilities and corner-case failures in large-scale SoC designs, we developed an AI-assisted risk prediction engine that analyzes heterogeneous historical development data and automatically derives instance-level and interface-level risk scores. The engine consumes bug reports, issue-tracking logs, RTL/spec change history, functional coverage trends, and design discussion emails, and converts them into unified multi-modal representations aligned with the IP-XACT instance hierarchy used in the PSS modeling database.

1) Data Acquisition and Pre-Processing

Four categories of inputs are integrated:

(a) Bug DB records, encoded with BERT[7] embeddings and clustered by failure mode;

(b) Issue/spec/RTL change logs, embedded with CodeBERT[8] to capture structural modification patterns;

(c) Email and engineering discussion logs, processed using topic modeling and sentiment analysis to detect timing or interface-related risk signals;

(d) IP revision/change lists, analyzed to identify high-churn modules and protocol updates.

Each data source is normalized and mapped to its corresponding IP-XACT instance ID (inst_id) to allow consistent multi-domain alignment.

2) Data Acquisition and Pre-Processing

A transformer-based multi-modal fusion encoder integrates text features, code diffs, structural IP-XACT attributes, and verification metrics into a unified risk embedding:

The final risk score is derived from three sub-components:

(a) Failure Likelihood Model, which predicts recurrence risk using severity embeddings and historical failure clusters ($F1 \approx 0.78$);

(b) Impact Propagation Model, which applies a GNN[9] on the IP-XACT connection graph to estimate how risks spread across dependent modules;

(c) Corner-Case Vulnerability Model, which detects action-sequence gaps and protocol-level untested behaviors using ML-based PSS action sequence modeling techniques (DVCon 2022).

3) Integration with PSS Modeling

Each instance receives a risk tag (0–3), which influences scenario weighting, action prioritization, and coverage guidance. Interfaces with elevated risk, such as blk_cpu.axi_m, trigger targeted stimulus generation involving AXI interleaving, boundary-crossing bursts, reset-in-flight, and timing-critical sequences, as illustrated in Figure 19.

A major contribution of this work is that AI predictions are directly integrated into the PSS modeling database, without requiring the creation of any additional AI-based test-generation models. As PSS is a modeling language, AI-guided updates only require adjusting instance metadata and constraint weights—not regenerating testbench infrastructure.

In other words, because PSS already provides a complete executable model that can be directly utilized by the AI engine, AI-based test generation can be achieved without developing numerous AI models trained on large-scale datasets, which significantly reduces the overhead typically associated with AI-driven verification automation.

Risk Assessment result	Generated PSS action sequence

{ "inst_id": "A1",
"risk_score": 3,
"dominant_factors": [
"High RTL churn in address decoder",
"Unresolved cluster: timing-sensitive stall bug pattern",
"Under-tested AXI4 interleaving sequences"],
"recommended_pss_actions": [
"axi_interleave_seq",
"burst_boundary_crossing",
"reset_mid_transaction"] }
	if (inst_db[A1].risk_score >= 2) {
	scenario.apply_priority("high");
	scenario.inject_action_seq(risky_action_list);
	}

Figure 19. Result of Risk prediction

Through the IP-XACT based PSS modeling database, we were able to identify the functionality and structural paths of each instance and automatically generate all associated tests. This capability indicates that the full set of functional cases, derived from all design paths and behavior relationships within the SoC, can be systematically uncovered through design-information analysis. As a result, corner-case conditions that previously appeared only intermittently at the silicon level can now be captured in the pre-silicon verification environment. This demonstrates the practical feasibility of achieving true shift-left SoC verification.

IV. CONCLUSION

As modern SoCs continue to increase in integration density and functional complexity, unexpected functional corner-case bugs are increasingly discovered at the silicon stage rather than during pre-silicon verification. Consequently, the reliability of functional coverage in design verification has diminished, and SoC verification is now often perceived not merely as functional verification but as integration verification. One contributing factor is that, despite the growing complexity of SoC designs, verification methodologies in many organizations still remain centered around UVM. Although PSS was introduced in 2018 as a successor methodology, its adoption as a mainstream approach has been limited.

We believe that many of the corner-case bugs that require days of testing to reproduce on silicon are not inherently silicon-only issues but rather the result of unanticipated functional conditions that could have been uncovered during simulation-based pre-silicon verification if sufficient functional coverage had been achieved. Achieving this requires identifying a broader range of exceptional scenarios and generating a significantly larger set of targeted tests. Over several years, we have pursued this challenge and demonstrated that combining PSS with IP-XACT can dramatically expand functional coverage, enabling the detection of bugs at the pre-silicon stage that would otherwise surface only in silicon.

PSS represents a key methodology for advancing shift-left SoC verification by enabling substantial functional coverage expansion, and its value will become even more significant in the emerging era of AI-driven verification. Although the examples presented in this paper focus on a simplified SoC, we hope that the concepts and techniques provided here will offer practical insights to engineers and EDA tool developers seeking to broaden the adoption and application of PSS.

REFERENCES

[1] Accellera Systems Initiative, Portable Test and Stimulus Standard (PSS) 3.0, 2025.
[2] IEEE Std 1685-2022, Standard for IP-XACT: Standard Structure for Packaging, Integrating, and Re-Using IP within Tools, IEEE, 2022.
[3] P. Karppa, L. Matilaninen, M. Ballance, “Building Portable Stimulus Into Your IP-XACT Flow” Proc. DVCon US, 2018
[4] M. Jang, W. Hyun, H. Ahn “PSS Action Sequence Modeling using Machine Learning” Proc. DVCon US, 2022
[5] M. Jang, S. Roe, Y. Kim “Early Detection of Functional Corner Case Bugs using Methodologies of the ISO 26262” Proc. DVCon US, 2023
[6] ISO, ISO 26262: Road Vehicles—Functional Safety, 2018.
[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Proc. NAACL-HLT, pp. 4171–4186, 2019.
[8] Z. Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Proc. EMNLP, pp. 1536–1547, 2020.
[9] J. Zhou et al., “Graph Neural Networks: A Review of Methods and Applications,” IEEE Transactions on Artificial Intelligence, vol. 1, no. 1, pp. 3–31, 2020.

APPENDIX A. EXAMPLE OF PSS MODELING DATABASE (JSON)


```
"instances": {
    "blk_cpu": {
        "id": "A",
        "rtl_path": "top.soc.blk_cpu",
        "instances": {
            "cpu0": {
                "id": "A1",
                "vendor": "arm.com",
                "library": "CortexA",
                "name": "XXXXX",
                "version": "1.0"
                "rtl_path": "top.soc.blk_cpu.cpu0",
                "interfaces": {
                    "axi_m": {
                        "id": "A1_IF1",
                        "protocol": "AXI4",
                        "role": "master",
                        "rtl_path": "top.soc.blk_cpu.cpu0",
                        "behaviors": {
                            "AW": {
                                "name": "axi_write_address",
                                "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.AW"          },
                            "W": {
                                "name": "axi_write_data",
                                "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.W"          },
                            "AR": {
                                "name": "axi_read_address",
                                "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.AR"          },
                            "R": {
                                "name": "axi_read_data",
                                "rtl_path": "top.soc.blk_cpu.cpu0.axi_m.R"          },
                            "B": {
```

```
"name": "axi_write_response",
"rtl_path": "top.soc.blk_cpu.cpu0.axi_m.B"    }   }
} },
"signals": {
  "state": {
    "id": "A1_SIG1",
    "name": "state",
    "dir": "out",
    "width": 2,
    "rtl_path": "top.soc.blk_cpu.cpu0.state",
    "behaviors": {
      "0": "idle",
      "1": "write",
      "2": "read"   } }  } },
"axi2acel": {
  "id": "A2",
  “vendor”: “samsung.com”,
  “library”: “bridge”,
  “name”: “axi2acel”,
  “version”: ”r0p0_0”
  "rtl_path": "top.soc.blk_cpu.axi2acel",
  "interfaces": {
    "axi_s": {
      "id": "A2_IF1",
      "protocol": "AXI4",
      "role": "slave",
      "rtl_path": "top.soc.blk_cpu.axi2acel"    },
    "acel_m": {
      "id": "A2_IF2",
      "protocol": "ACE-Lite",
      "role": "master",
      "rtl_path": "top.soc.blk_cpu.axi2acel"   }   }   }   },
  "signals": {
    "o_state": {
    "id": "A_SIG1",
    "name": "o_state",
    "dir": "out",
    "width": 2,
    "rtl_path": "top.soc.blk_cpu.o_state"  }  }  },
"blk_nocl0": {
  "id": "B",
  "rtl_path": "top.soc.blk_nocl0",
  "interfaces": {
    "acel_s": {
      "id": "B_IF1",
      "protocol": "ACE-Lite",
      "role": "slave",
      "rtl_path": "top.soc.blk_nocl0"    },
    "axi_m": {
      "id": "B_IF2",
      "protocol": "AXI4",
      "role": "master",
      "rtl_path": "top.soc.blk_nocl0"   }
  }
  "dst_states": [
    {
      "name": "idle",
      "code": 0,
      "description": "No outstanding transactions; ready to accept a new AXI write."    },
    {
```

```
        "name": "busy",
        "code": 1,
        "description": "Internal buffer is partially/full; write latency may increase."    },
      {
        "name": "clock_gated",
        "code": 2,
        "description": "Clock gating is enabled; AXI write must re-enable clock before data is accepted."    },
      {
        "name": "power_gated",
        "code": 3,
        "description": "Block is powered down; AXI write is blocked or routed as an error."    },
      {
        "name": "error",
        "code": 4,
        "description": "Error condition such as address decode error or protection violation."    },
      {
        "name": "recovering",
        "code": 5,
        "description": "Block is recovering from an error and may temporarily stall or drop transactions."    }
    ]
  },
  "blk_mif": {
    "id": "C",
    "rtl_path": "top.soc.blk_mif",
    "interfaces": {
      "axi_s": {
        "id": "C_IF1",
        "protocol": "AXI4",
        "role": "slave",
        "rtl_path": "top.soc.blk_mif"   }
    }
    "dst_states": [
      {
        "name": "idle",
        "code": 0,
        "description": "No outstanding transactions; ready to accept a new AXI write."    },
      {
        "name": "busy",
        "code": 1,
        "description": "Internal buffer is partially/full; write latency may increase."    },
      {
        "name": "clock_gated",
        "code": 2,
        "description": "Clock gating is enabled; AXI write must re-enable clock before data is accepted."    },
      {
        "name": "power_gated",
        "code": 3,
        "description": "Block is powered down; AXI write is blocked or routed as an error."    },
      {
        "name": "error",
        "code": 4,
        "description": "Error condition such as address decode error or protection violation."    },
      {
        "name": "recovering",
        "code": 5,
        "description": "Block is recovering from an error and may temporarily stall or drop transactions."    }
    ]
  }
},

  "connections": [
    {
      "id": "CONN_01",
      "kind": "interface",
      "protocol": "AXI4",
      "src": { "inst_id": "A1", "if_id": "A1_IF1" },
      "dst": { "inst_id": "A2", "if_id": "A2_IF1" }   },
    {
```

```
      "id": "CONN_02",
      "kind": "interface",
      "protocol": "ACE-Lite",
      "src": { "inst_id": "A2", "if_id": "A2_IF2" },
      "dst": { "inst_id": "B0", "if_id": "B0_IF1" }      },
     {
      "id": "CONN_03",
      "kind": "interface",
      "protocol": "AXI4",
      "src": { "inst_id": "B0", "if_id": "B0_IF2" },
      "dst": { "inst_id": "C0", "if_id": "C0_IF1" }      },
     {
      "id": "CONN_04",
      "kind": "signal",
      "src": { "inst_id": "A1", "sig_id": "A1_SIG1" },
      "dst": { "inst_id": "A0", "sig_id": "A0_SIG1" },
      "description": "cpu0.state[1:0] drives blk_cpu.o_state"      }
  ],

  "path_catalog": [
     {
      "path_id": "PATH_01",
      "src": { "inst_id": "A1", "if_id": "A1_IF1" },
      "dst": { "inst_id": "C0", "if_id": "C0_IF1" },
      "route": [
        { "hop_inst": "A2", "via_if": ["A2_IF1", "A2_IF2"] },
        { "hop_inst": "B0", "via_if": ["B0_IF1", "B0_IF2"] }        ],
      "protocol_chain": ["AXI4", "ACE-Lite", "AXI4"],
      "rtl_hierarchy": [
       "top.soc.blk_cpu.cpu0",
       "top.soc.blk_cpu.axi2acel",
       "top.soc.blk_nocl0",
       "top.soc.blk_mif"       ]     }
  ]
```


APPENDIX B. EXAMPLE OF GENERATED PSS MODEL

A. *db_types.pss*

```
package db_types {

  // Common Types and Constants
  type design_id_t : string;
  type block_id_t  : string;
  type inst_id_t   : string;
  type if_id_t     : string;
  type sig_id_t    : string;
  type proto_t     : string;
  type role_t      : string;

  struct addr_rng_t {
    uint64 lo;
    uint64 hi;
  };

  // Maximum Array Sizes Within a Path
  const MAX_BEHAVIORS    = 8;
  const MAX_SIG_LINKS    = 16;
  const MAX_HOPS         = 8;
  const MAX_VIA_IF       = 4;
  const MAX_PROTO_CHAIN  = 8;
  const MAX_RTL_NODES    = 16;
  const MAX_ALLOW_RANGES = 8;

  // Record Types
  // Instance
  struct inst_rec_t {
    inst_id_t  id;
```

```
    string     name;
    string     type;
    string     rtl_abs;
    block_id_t owner_blk;
    inst_id_t  parent;
    int        depth;
  };

  // Interface(Port)
  struct if_rec_t {
    if_id_t    id;
    inst_id_t  parent_inst;
    string     name;
    proto_t    proto;
    role_t     role;
    string     rtl_abs;
  };

// Signal and Behavior Labels
  struct sig_behavior_t {
    int    value;
    string label;
  };

  struct sig_rec_t {
    sig_id_t       id;
    inst_id_t      parent_inst;
    string         name;
    string         dir;
    int            width;
    string         rtl_abs;
    list<sig_behavior_t> behaviors;
  };
   typedef struct {
       string   kind;              // "SIG" or "IF"
       sig_id_t sig_id;
       if_id_t  if_id;
       int      sig_val;           // for signal behavior
       string   if_bkey;           // for interface behavior (AW/W/AR/R/B)
       string   action_name; // e.g. "cpu::cpu_write32"
  } behavior_action_t;

  // Block-internal Signal Connections
  struct sig_link_t {
    sig_id_t   src_sig;
    sig_id_t   dst_sig;
    string     description;
  };

  // Hop Entry in a Path
  struct hop_rec_t {
    inst_id_t  hop_inst;
    list<if_id_t> via_if;
  };

  // Top-Level Path Record
  struct path_rec_t {
    string          path_id;
    if_id_t         src_if;
    if_id_t         dst_if;
    list<hop_rec_t>  route;
    list<string>     protocol_chain;
    list<string>     rtl_hierarchy;
    list<addr_rng_t> allow;
  };
};
```

B. blk_cpu_db.pss

```
import db_types::*;

package blk_cpu_db {

    behavior_action_t behavior_actions[128];
    int behavior_actions_n = 0;

    function void add_sig_behavior_action( sig_id_t sig_id, int  sig_val, string   action_name );
         behavior_action_t e;
         e.kind       = "SIG";
```

```
        e.sig_id      = sig_id;
        e.sig_val     = sig_val;
        e.if_id       = "";
        e.if_bkey     = "";
        e.action_name = action_name;

        behavior_actions[behavior_actions_n] = e;
        behavior_actions_n++;
    endfunction

    function void add_if_behavior_action( if_id_t if_id, string  if_bkey, string  action_name );
        behavior_action_t e;
        e.kind        = "IF";
        e.if_id       = if_id;
        e.if_bkey     = if_bkey;
        e.sig_id      = "";
        e.sig_val     = -1;
        e.action_name = action_name;

        behavior_actions[behavior_actions_n] = e;
        behavior_actions_n++;
    endfunction

    action find_actions_by_behavior {
      input  string   kind;        // "SIG" or "IF"
      input  sig_id_t sig_id;
      input  if_id_t  if_id;
      input  int      sig_val;
      input  string   if_bkey;
      output string   actions[16];
      output int      actions_n;

      exec body {
        int i;
        actions_n = 0;

        for (i = 0; i < behavior_actions_n; i++) begin
            if (behavior_actions[i].kind != kind)
                continue;
            if (kind == "SIG") begin
                if (behavior_actions[i].sig_id  == sig_id &&
                behavior_actions[i].sig_val == sig_val) begin
                    if (actions_n < 16) begin
                        actions[actions_n] = behavior_actions[i].action_name;
                        actions_n++;
                    end
                end
            end
            if (kind == "IF") begin
                if (behavior_actions[i].if_id   == if_id &&
                behavior_actions[i].if_bkey == if_bkey) begin
                    if (actions_n < 16) begin
                        actions[actions_n] = behavior_actions[i].action_name;
                        actions_n++;
                    end
                end
            end
        end
      }
  }

   component blk_cpu_db_c {

   const block_id_t BLK = "blk_cpu";

   // Inventory Size
   const int N_INSTS = 3;
   const int N_IFS   = 3;
   const int N_SIGS  = 2;
   const int N_LINKS = 1;

   // Inventory Table
   array<inst_rec_t, N_INSTS> insts;
   array<if_rec_t,   N_IFS>   ifs;
   array<sig_rec_t,  N_SIGS>  sigs;
```

```
array<sig_link_t, N_LINKS> sig_links;

// Data Population
action build_db {
  exec body {
    // ---- instances ----
    insts[0].id="A0";
    insts[0].name="blk_cpu";
    insts[0].type="BLOCK";
    insts[0].rtl_abs="top.soc.blk_cpu";
    insts[0].owner_blk=BLK;
    insts[0].parent="T0";
    insts[0].depth=1;

    insts[1].id="A1";
    insts[1].name="cpu0";
    insts[1].type="RISC_CORE";
    insts[1].rtl_abs="top.soc.blk_cpu.cpu0";
    insts[1].owner_blk=BLK;
    insts[1].parent="A0";
    insts[1].depth=2;

    insts[2].id="A2";
    insts[2].name="axi2acel";
    insts[2].type="AXI2ACE_BRIDGE";
    insts[2].rtl_abs="top.soc.blk_cpu.axi2acel";
    insts[2].owner_blk=BLK;
    insts[2].parent="A0";
    insts[2].depth=2;

    // ---- interfaces ----
    ifs[0].id="A1_IF1";
    ifs[0].parent_inst="A1";
    ifs[0].name="axi_m";
    ifs[0].proto="AXI4";
    ifs[0].role="master";
    ifs[0].rtl_abs="top.soc.blk_cpu.cpu0";

    ifs[1].id="A2_IF1";
    ifs[1].parent_inst="A2";
    ifs[1].name="axi_s";
    ifs[1].proto="AXI4";
    ifs[1].role="slave";
    ifs[1].rtl_abs="top.soc.blk_cpu.axi2acel";

    ifs[2].id="A2_IF2";
    ifs[2].parent_inst="A2";
    ifs[2].name="acel_m";
    ifs[2].proto="ACE-Lite";
    ifs[2].role="master";
    ifs[2].rtl_abs="top.soc.blk_cpu.axi2acel";

    // ---- signals ----
    // cpu0.state[1:0]
    sigs[0].id="A1_SIG1";
    sigs[0].parent_inst="A1";
    sigs[0].name="state";
    sigs[0].dir="out";
    sigs[0].width=2;
    sigs[0].rtl_abs="top.soc.blk_cpu.cpu0.state";
    sigs[0].behaviors_n = 3;
    sigs[0].behaviors[0].value=0;
    sigs[0].behaviors[0].label="idle";
    sigs[0].behaviors[1].value=1;
    sigs[0].behaviors[1].label="write";
    sigs[0].behaviors[2].value=2;
    sigs[0].behaviors[2].label="read";

    // blk_cpu.o_state[1:0]
    sigs[1].id="A0_SIG1";
    sigs[1].parent_inst="A0";
    sigs[1].name="o_state";
    sigs[1].dir="out";
    sigs[1].width=2;
    sigs[1].rtl_abs="top.soc.blk_cpu.o_state";
    sigs[1].behaviors_n = 0;

    // ---- local signal link: cpu0.state -> blk_cpu.o_state ----
    sig_links[0].src_sig="A1_SIG1";
    sig_links[0].dst_sig="A0_SIG1";
```

```
        sig_links[0].description="cpu0.state[1:0] drives blk_cpu.o_state";      }    }   }  };
};

// Lookup Utilities
   // Return the Full Table
   action list_insts {
     output array<inst_rec_t, N_INSTS> out;
     output int n;
     exec body {
       n=N_INSTS;
       for (int i=0;i<N_INSTS;i++) out[i]=insts[i];
     }
   }

   action list_ifs {
     output array<if_rec_t, N_IFS> out;
     output int n;
     exec body {
       n=N_IFS;
       for (int i=0;i<N_IFS;i++) out[i]=ifs[i];
     }
   }

   action list_sigs {
     output array<sig_rec_t, N_SIGS> out;
     output int n;
     exec body {
       n=N_SIGS;
       for (int i=0;i<N_SIGS;i++) out[i]=sigs[i];
     }
   }

   // Lookup by Specific Instance, Port, or Signal
   action inst_by_id {
     input inst_id_t id;
     output inst_rec_t rec;
     output bit found;
     exec body {
       found=0;
       for (int i=0;i<N_INSTS;i++) {
         if (insts[i].id==id){ rec=insts[i]; found=1; break; }
       }
     }
   }

   action ifs_by_inst {
     input inst_id_t id;
     output array<if_rec_t, N_IFS> out;
     output int n;
     exec body {
       n=0;
       for (int i=0;i<N_IFS;i++) {
         if (ifs[i].parent_inst==id){ out[n]=ifs[i]; n++; }
       }
     }
   }

   action sigs_by_inst {
     input inst_id_t id;
     output array<sig_rec_t, N_SIGS> out;
     output int n;
     exec body {
       n=0;
       for (int i=0;i<N_SIGS;i++) {
         if (sigs[i].parent_inst==id){ out[n]=sigs[i]; n++; }
       }
     }
   }

  action local_sig_links_via {
     input sig_id_t sid;
     output array<sig_link_t, N_LINKS> out;
     output int n;
     exec body {
       n=0;
       for (int i=0;i<N_LINKS;i++) {
         if (sig_links[i].src_sig==sid || sig_links[i].dst_sig==sid) {
           out[n]=sig_links[i]; n++;
         }
       }
```

```
    }
```

C. *cpu.pss*

```
import blk_cpu_db;

package cpu {

  const inst_id_t CPU0_ID        = "A1";
  const if_id_t   CPU0_AXI_M_IF  = "A1_IF1";
  const sig_id_t  CPU0_STATE_SIG = "A1_SIG1";

  const int STATE_IDLE  = 0;
  const int STATE_WRITE = 1;
  const int STATE_READ  = 2;

  action cpu_write32 {
    input uint64 addr;
    input uint32 data;
    activity { …}
  }

  action cpu_read32 {
    input uint64 addr;
    output uint32 data;
    activity { … }
  }

  action go_sleep {
    activity { … }
  }

  action register_cpu_actions {

    exec body {
      // cpu_write32
      blk_cpu_db::add_sig_behavior_action(
        CPU0_STATE_SIG, STATE_WRITE, "cpu::cpu_write32"
      );

      blk_cpu_db::add_if_behavior_action(
        CPU0_AXI_M_IF, "AW", "cpu::cpu_write32"
      );
      blk_cpu_db::add_if_behavior_action(
        CPU0_AXI_M_IF, "W",  "cpu::cpu_write32"
      );
      blk_cpu_db::add_if_behavior_action(
        CPU0_AXI_M_IF, "B",  "cpu::cpu_write32"
      );

      // cpu_read32
      blk_cpu_db::add_sig_behavior_action(
        CPU0_STATE_SIG, STATE_READ, "cpu::cpu_read32"
      );

      blk_cpu_db::add_if_behavior_action(
        CPU0_AXI_M_IF, "AR", "cpu::cpu_read32"
      );
      blk_cpu_db::add_if_behavior_action(
        CPU0_AXI_M_IF, "R",  "cpu::cpu_read32"
      );

      // go_sleep
      blk_cpu_db::add_sig_behavior_action(
        CPU0_STATE_SIG, STATE_IDLE, "cpu::go_sleep"
      );
    }
  }
```

```
}
```

D. top_db.pss

```
import db_types;

package top_db;

component path_catalog_c {
  const int N_PATHS = 1;
  path_rec_t paths[N_PATHS];

  action build_db {
    exec body {
      // PATH_01 : A1_IF1 → (A2_IF1,A2_IF2) → (B0_IF1,B0_IF2) → C0_IF1
      paths[0].path_id = "PATH_01";
      paths[0].src_if  = "A1_IF1";
      paths[0].dst_if  = "C0_IF1";

      // route
      paths[0].route_n = 2;

      paths[0].route[0].hop_inst = "A2";
      paths[0].route[0].via_if_n = 2;
      paths[0].route[0].via_if[0] = "A2_IF1";
      paths[0].route[0].via_if[1] = "A2_IF2";

      paths[0].route[1].hop_inst = "B0";
      paths[0].route[1].via_if_n = 2;
      paths[0].route[1].via_if[0] = "B0_IF1";
      paths[0].route[1].via_if[1] = "B0_IF2";

      // protocol_chain
      paths[0].protocol_chain_n = 3;
      paths[0].protocol_chain[0] = "AXI4";
      paths[0].protocol_chain[1] = "ACE-Lite";
      paths[0].protocol_chain[2] = "AXI4";

      // rtl_hierarchy
      paths[0].rtl_hierarchy_n = 4;
      paths[0].rtl_hierarchy[0] = "top.soc.blk_cpu.cpu0";
      paths[0].rtl_hierarchy[1] = "top.soc.blk_cpu.axi2acel";
      paths[0].rtl_hierarchy[2] = "top.soc.blk_nocl0";
      paths[0].rtl_hierarchy[3] = "top.soc.blk_mif";

      // Allow address range : none
      paths[0].allow_n = 0;
    }
  }

  // List of Paths that Must Traverse a Specific Interface
  action paths_via_if {
    input  if_id_t through;
    output path_rec_t out[];
    output int n;
    activity { build_db; }
    exec body {
      n = 0;
      for (int i=0;i<N_PATHS;i++) {
        bit hit = 0;
        if (paths[i].src_if == through || paths[i].dst_if == through) hit = 1;
        for (int h=0; h<paths[i].route_n && !hit; h++) {
          for (int k=0; k<paths[i].route[h].via_if_n; k++) {
            if (paths[i].route[h].via_if[k] == through) { hit = 1; break; }
          }
        }
        if (hit) { out[n] = paths[i]; n++; }
      }
    }
  }

  // Direct Path from src_if to dst_if
  action paths_src_to_dst {
    input  if_id_t src_if;
    input  if_id_t dst_if;
    output path_rec_t out[];
    output int n;
    activity { build_db; }
    exec body {
```

```
    n = 0;
    for (int i=0;i<N_PATHS;i++) if (paths[i].src_if==src_if && paths[i].dst_if==dst_if) { out[n]=paths[i]; n++; }
   }
  }
}

endpackage
```

E. test_top.pss

```
// test_top.pss
import db_types;
import blk_cpu_db;
import blk_nocl0_db;
import blk_mif_db;
import top_db;
import cpu;

component test_top {

  // DB handle
  blk_cpu_db::blk_cpu_db_c       CPU_DB;
  blk_nocl0_db::blk_nocl0_db_c   NOC_DB;
  blk_mif_db::blk_mif_db_c       MIF_DB;
  top_db::path_catalog_c         TOP;

  // Component Initialization: Database Loading
  exec init_up {
   activity {
      do CPU_DB.build_db;
      do NOC_DB.build_db;
      do MIF_DB.build_db;
      do TOP.build_db;
      do cpu::register_cpu_actions;
   }
  }

  // Test Entry Point
action entry {
    activity {

    }
  }
}
```