From Specification to Closure: A Semi-Automated Coverage-Driven Verification Methodology for Cache Coherent Home Nodes

Kavin Rajendran, Anishmon Soosai, Sumit Dhamanwala, Kranthi Konganti,
Shivaprasad Naranapura Chandrashekara Swamy
Openedges, Austin, USA

1. ABSTRACT

Achieving verification closure in modern cache-coherent Network-on-Chip (NoC) designs is a significant challenge, especially with the increasing complexity of protocols such as AMBA CHI. In these systems, coherence transactions initiated by a Requester Node (RN) can result in intricate flows involving multiple packers across Request, Snoop, Response, and Data channels often executed out-of-order. The Home Node (HN), being the coherence authority, must handle these interactions reliably while adhering to MOESI protocol semantics. This complexity creates a vast verification space where each opcode's lifecycle involves several potential permutations of state, channel flow, and agent interactions. Without a structured methodology to measure how well stimulus exercises the design, it is difficult to ensure meaningful verification progress or identify coverage gaps. Given the industry's aggressive time-to-market demands, relying solely on directed tests or randomized traffic is no longer viable unless coupled with robust, transaction-aware functional coverage. Functional coverage plays a pivotal role in this ecosystem. It not only provides quantitative confidence in design validation but also acts as a feedback loop to guide stimulus development. For a CHI-based Home Node, however, off-the-shelf coverage from third-party VIPs is often insufficient, as they tend to focus on protocol compliance rather than design-specific behaviors. This necessitates a custom, reusable, and scalable solution.

2. INTRODUCTION

This paper introduces the Coherent Opcode Flow Coverage Architecture, a layered, semi-automated methodology designed to address the unique verification challenges of CHI-based Home Node designs. The architecture is built on three key pillars:

2.1 Automated Coverpoint and Cross-Cover Extraction from Microarchitecture Specification

At the heart of this solution is an automation toolchain that converts a structured Microarchitecture specification of the HN-F (Home Node - Fully Coherent) design into actionable coverage definitions. A Python-based parser processes the specification and generates a JSON representation of relevant opcode flows, channel interactions, and MOESI state transitions. Using configurable user input, the tool identifies and generates meaningful coverpoints and cross-covers, mapping them to coherent lifecycle events. This automation significantly reduces manual coding effort, minimizes human error in translating the spec to coverage models, and makes the system robust to spec changes. A change in opcode behavior or allowed flow permutations can be updated in the spec, and the corresponding coverage updates are propagated automatically.

2.2 UVM-Based Opcode Lifecycle Tracking Using Scoreboard Integration

The second contribution is a SystemVerilog UVM methodology for constructing and sampling full opcode flows. Traditional testbenches often track packets on each channel (Req/Snp/Rsp/Dat) independently using per-channel monitors. Later, separate logic attempts to match corresponding transactions, introducing both complexity and latency. Instead, this architecture leverages the existing scoreboard, which already has visibility into complete transaction flows for checking correctness. When a transaction is ready to retire, the scoreboard constructs a coverage lifecycle object, encapsulating:

- The original request
- All associated snoop requests issued
- All Snoop responses and data packets received

This object is pushed via TLM ports to the coverage model, eliminating redundant tracking logic and ensuring atomicity of coverage sampling. This tightly coupled yet non-intrusive approach minimizes simulation overhead while maximizing accuracy and modularity.

2.3 Functional Coverage Model for Coherent Opcode Flows

The final piece of the architecture is a functional coverage model that unpacks and samples the lifecycle object. The model defines cover groups that capture:

- Opcode and transaction type (e.g., ReadShared, ReadUnique)
- Types and combinations of snoop responses received (e.g., SnpResp, SnpRespDataFwd)
- Response vs Forwarded cache states (e.g., SC, I_PD)
- Legal and illegal MOESI state transitions
- Cross-coverage between opcode class and snoop outcome
- Any HA internal state meta data (e.g., SF Hit/Miss, SLC Hit/Miss)

The model makes direct use of the auto-generated coverage code from the JSON input, ensuring alignment with spec-defined flows. Sampling is triggered only when a full and valid transaction flow has been observed, thereby avoiding false positives and improving **signal-to-noise ratio in coverage metrics**.

3. Methodology Process Flow

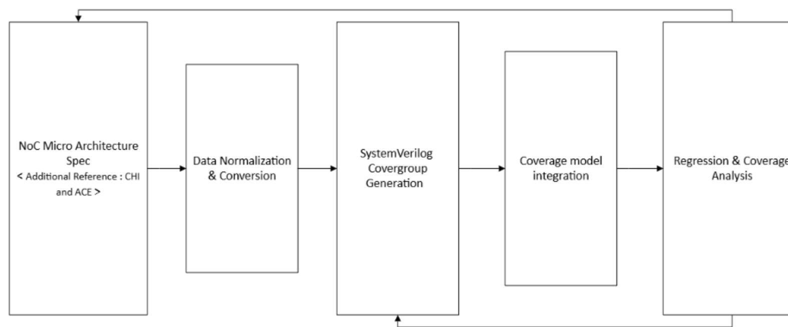

Figure 1. Process flow chart

3.1 Architecture Specification

To illustrate the extraction of relevant data from the micro-architecture specification, this paper will present a simplified implementation of how a Home Node (HN-F) handles a coherent request, specifically a ReadOnce transaction. The mechanics of this process are detailed below in both an FSM diagram and a state transition table.

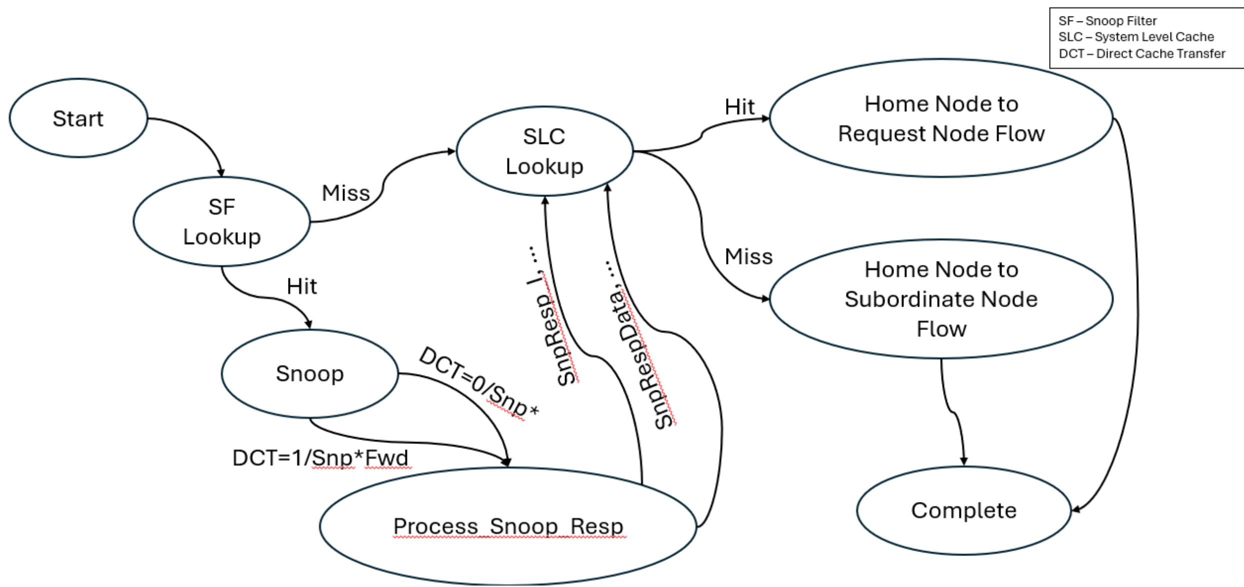

Figure 2. Home Node micro-architecture FSM

Expressing the Home Node's micro-architecture in an FSM table or diagram makes it conducive to parsing and allows it to serve as direct input to the automated flow.

Table 1 details the initial set of possible transitions once a ReadOnce request arrives at the Home Node. Upon being picked up for processing, a Snoop Filter lookup is performed. Based on the lookup's outcome (Hit or Miss), the next state transitions to either Snoop or SLC Lookup, respectively.

Opcode	Current State	Lookup Result	Next State
ReadOnce	SF Lookup	Hit	Snoop
		Miss	SLC Lookup

Table 1. FSM Table – Snoop Filter Lookup

Let's consider the case where the lookup results in a Hit, causing the FSM to transition to the Snoop state. In this state, the Home Node determines the appropriate snoop type for the request; for a ReadOnce transaction, this will be either SnpOnce or SnpOnceFwd.

Opcode	Current State	DCT Evaluation	Snoop Req	Next State
RdOnce	Snoop	0	SnpOnce	Process_Snoop_Resp
		1	SnpOnceFwd	Process_Snoop_Resp

Table 2. FSM Table - Snoop

After the snoop request is sent, the FSM moves to the process_snoop_response state to await responses. This state presents a major Home Node verification challenge due to the complexity of inputs: one out of five possible snoop responses can arrive via two possible channels (DAT and RSP). Crucially, the Resp and FwdState fields-each capable of holding five different state encodings-determine the next state, causing the decision space to increase exponentially.

Opcode	Current State	SnpResp_<Resp>	SnpRespData_<Resp>	SnpRespDataPtl_<Resp>	SnpResp_<Resp>_Fwd_<FwdState>	SnpRespData_<Resp>_Fwd_<FwdState>	Next State
RdOnce	Process_snoop_response	I/SC/UC					SLC Lookup
			I/SC/UC/SD/UD				SLC Lookup
				UD/I_PD			SLC Lookup
					I,I/SC,I/UC,I/SD,I/UD,I		SLC Lookup
						I_PD,I/SC_PD,I	SLC Lookup

Table 3. FSM Table – Process Snoop Response

3.2 Data Normalization & Conversion

To make the FSM tables from micro architecture easy to process, the tables are reviewed for any text or notations used that may need to be eliminated or reworded for clarity. This step ensures that all data input points follow a standardized format and are represented in a script accessible format like xls or csv, which is essential for automation. The process is semi-automated but includes manual checks to confirm that the generated xls files are accurate and complete. These checks help catch any formatting issues or missing fields early in the flow.

After normalization, a python-based converter script is implemented that reads the FSM tables(.xls) and converts them into JSON format. JSON is chosen because it is structured, easy to parse, and works well for automation. The input xls file now contains important fields such as opcodes, responses, and cross-combinations. These fields are grouped and stored in JSON, which becomes the foundation for the next steps. Below is an example snippet of the generated JSON structure:

```json
[
  {
    "Opcode": "ReadOnly",
    "Current State": "SF Lookup",
    "Lookup Result": "Hit",
    "Next State": "Snoop"
  },
  {
    "Opcode": "ReadOnly",
    "Current State": "SF Lookup",
    "Lookup Result": "Miss",
    "Next State": "SLC Lookup"
  }
]
```

Snippet 1. JSON representation of Table 1.

```json
[
  {
    "Opcode": "RdOnce",
    "Current State": "Snoop",
    "DCT Evaluation": "0",
    "Snoop Req": "SnpOnce",
    "Next State": "Process_Snoop_Resp"
  },
  {
    "Opcode": "RdOnce",
    "Current State": "Snoop",
    "DCT Evaluation": "1",
    "Snoop Req": "SnpOnceFwd",
    "Next State": "Process_Snoop_Resp"
  }
]
```

Snippet 2. JSON representation of Table 2.

```json
[
  {
    "Opcode": "RdOnce",
    "Current State": "Process_snoop_response",
    "SnpResp_<Resp>": "I/SC/UC",
    "SnpRespData_<Resp>": "",
    "SnpRespDataPtl_<Resp>": "",
    "SnpResp_<Resp>_Fwd_<FwdState>": "",
    "SnpRespData_<Resp>_Fwd_<FwdState>": "",
    "Next State": "SLC Lookup"
  },
  {
    "Opcode": "RdOnce",
    "Current State": "Process_snoop_response",
    "SnpResp_<Resp>": "",
    "SnpRespData_<Resp>": "I/SC/UC/CSD/UD",
    "SnpRespDataPtl_<Resp>": "",
    "SnpResp_<Resp>_Fwd_<FwdState>": "",
    "SnpRespData_<Resp>_Fwd_<FwdState>": "",
    "Next State": "SLC Lookup"
  },
  {
    "Opcode": "RdOnce",
    "Current State": "Process_snoop_response",
    "SnpResp_<Resp>": "",
    "SnpRespData_<Resp>": "",
    "SnpRespDataPtl_<Resp>": "UD/I_PD",
    "SnpResp_<Resp>_Fwd_<FwdState>": "",
    "SnpRespData_<Resp>_Fwd_<FwdState>": "",
    "Next State": "SLC Lookup"
  },
  ...
  ...
]
```

Snippet 3. JSON representation of Table 3.

This structured format not only simplifies coverage generation but also makes the data reusable for other verification tasks like stimulus generation and consistency checks.

3.3 Generating SystemVerilog Covergroups

Next, the JSON data is used to create SystemVerilog cover groups. The grouping can be based on any relevant design functionality, in this example it is based on opcodes, responses, and their cross-combinations, with a strong focus on cross-coverage between a request, its snoop request and the various possible snoop response types and states it can receive. This automation ensures that all possible cross-combinations are generated according to the design requirements.

To make the cover groups easier to read and maintain, additional labeling and alignment are applied. Debug hooks are also added to help identify and analyze coverage gaps during simulation. These hooks provide valuable insights when debugging complex scenarios where coverage points are not being hit.

To monitor unsupported state combinations, illegal bins are generated for response states and forward states not valid for a given flow. The converter script makes this generation optional, allowing the coverage model to serve a dual purpose as both a coverage metric and a correctness checker.

```
covergroup cg_ReadOnce;
  option.per_instance = 1;
  option.goal = 100;

  cp_SLC_LkUp: coverpoint slc_lkup_status_fcov {
        bins ValidCom[]        = { MISS ,HIT};
    }

  cp_SLC_VAR : coverpoint HA_VAR{
        bins  HA_VAR       = { ABC, DEF};// Picked dummy value for demo
    }

  cp_RdOnce_SnpResp_RespState: coverpoint cov_snp_rsp_pkt.Resp {
        bins ValidCom[]        = { I, SC, UC };
        illegal_bins  illegal[] = { I_PD, SC_PD, SD, SD_PD, UC_PD };
    }
  cp_RdOnce_SnpResp_RespState_slc_opcode : cross cp_RdOnce_SnpResp_RespState,cp_SLC_LkUp;

  cp_RdOnce_SnpRespDataFwd_RespState : coverpoint cov_snp_rsp_data_pkt.Resp {
    bins RespData_I_PD  = {I_PD};
    bins RespData_SC_PD = {SC_PD};
    illegal_bins  illegal[] = {UC_PD,SD_PD};
  }

  cp_RdOnce_SnpRespDataFwd_FwdState : coverpoint cov_snp_rsp_data_pkt.FwdState {
    bins Fwd_I     = { I };
    illegal_bins  illegal[] = {I_PD,SD_PD,SD,SC_PD,UC};
  }

  cross cp_RdOnce_SnpRespDataFwd_RespState, cp_RdOnce_SnpRespDataFwd_FwdState{
    bins cross_SnpResp_I_PD_SnpRespFwd_I     = binsof(cp_RdOnce_SnpRespDataFwd_RespState.RespData_I_PD)  && binsof(cp_RdOnce_SnpRespDataFwd_FwdState.Fwd_I);
    bins cross_SnpResp_SC_PD_SnpRespFwd_I    = binsof(cp_RdOnce_SnpRespDataFwd_RespState.RespData_SC_PD) && binsof(cp_RdOnce_SnpRespDataFwd_FwdState.Fwd_I);
  }

 CP_RDONCE_SNPRESPDATAFWD_RESPSTATE_FWDSATE_HA_OPCODE : cross cp_RdOnce_SnpRespDataFwd_RespState,cp_RdOnce_SnpRespDataFwd_FwdState,cp_SLC_VAR;
<.....>
endgroup : cg_ReadOnce
```

Snippet 4. Auto generated system Verilog cover group.

The code snippet above shows what the output of the JSON to SV cover group converter script looks like for ReadOnce opcode and its properties.

3.4 Coverage Model Integration

Once the cover groups are generated, they are integrated into the Network-on-Chip (NoC) verification environment. The connection between the scoreboard and the coverage model is established using a TLM Analysis port. Coverage sampling is triggered when the scoreboard deallocates a packet, which means the flow for the given opcode is complete and the packet contains all relevant details (request, snoops and responses).

This methodology supports two types of coverage collection models:

Many-to-One Mapping: Multiple HNF scoreboards feeding into a single coverage model.

One-to-One Mapping: Each HNF scoreboard connected to its own coverage model.

Manual reviews are performed to ensure that the generated cover groups match the micro-architecture specification.

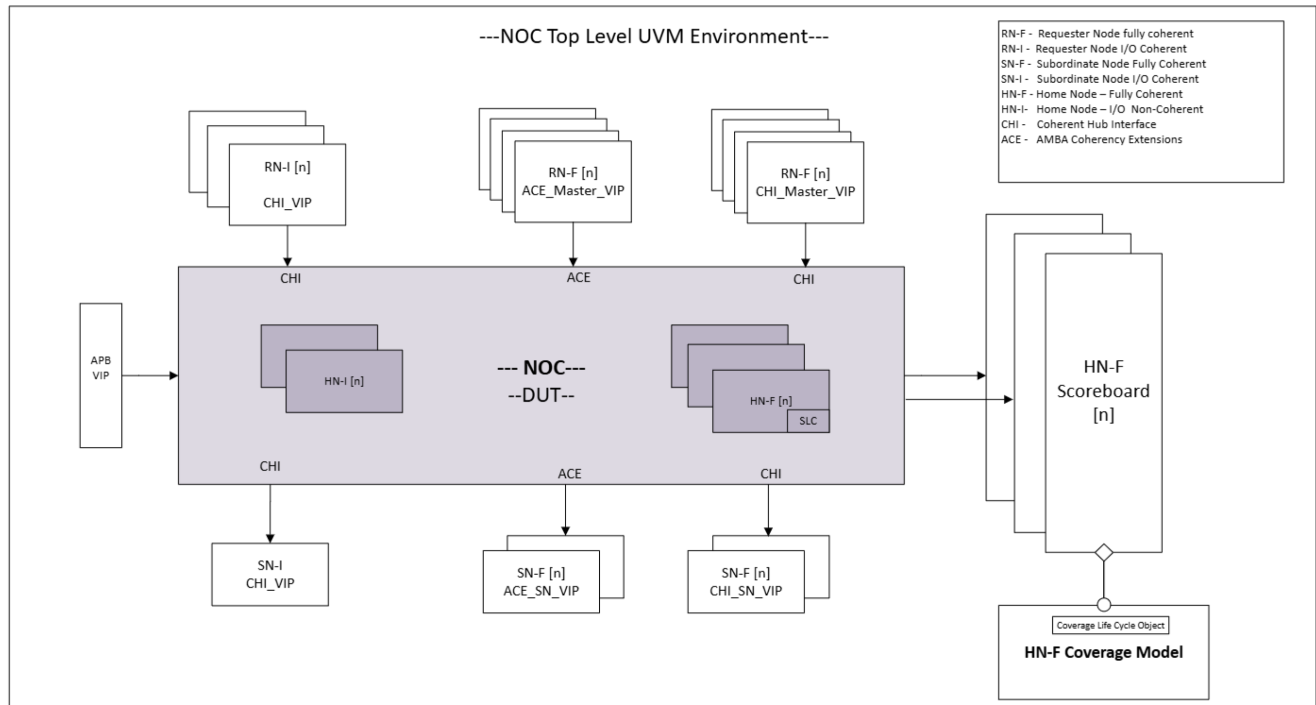

Figure 3. NoC UVM Environment

3.4.1 Scoreboard

The proposed methodology relies on a per instance (home node) scoreboard architecture. In this architecture, the scoreboard tracks the complete lifecycle of request as it comes into a home node, encompassing the entire sequence of an opcode flow which begins with the request, followed by any snoops and concludes with any relevant response and data transactions. Once the scoreboard receives a request packet, it allocates a unique entry for the corresponding transaction and continuously monitors for the transactions that are expected out of the DUT. When the scoreboard matches an expected transaction with an actual transaction, it's stored in a buffer until all expected packets have been received or transmitted by Home Node. Once an opcode flow completes which is marked by matching the last expected transaction from the home node for that request, the scoreboard copies all relevant data points for this flow into an opcode lifecycle coverage object and uses TLM port to write the coverage object to any subscriber component, specifically a coverage model in this case. The scoreboard then proceeds to deallocate the specific entry in its buffer.

As explained above, the process involves directly utilizing scoreboard data, so it naturally depends on the correctness of the scoreboard. As in typical UVM based environments the DUT and scoreboard hold each other responsible for being correct and any deviance of one of these components would result in flagging a mismatch error but there might be cases where both these components are doing something incorrect in tandem which might go unnoticed and as a result provide incorrect data for coverage sampling. To address this issue, we conducted thorough and periodical reviews of the coverage collection and sampling procedures, which confirmed that the observed coverage data is aligned with the architectural requirements.

There are two types of coverage model integrations implemented. The first involves multiple HNF scoreboards feeding into a single coverage model, where the coverage objects from all HNF scoreboards in the env are sent to a centralized coverage model. The second method connects each instance of HNF scoreboard to its own coverage model, resulting in a one-to-one connection. However, this approach requires more time to achieve complete functional coverage but gives coverage data on a per instance basis for designs that have different HNF configurations.

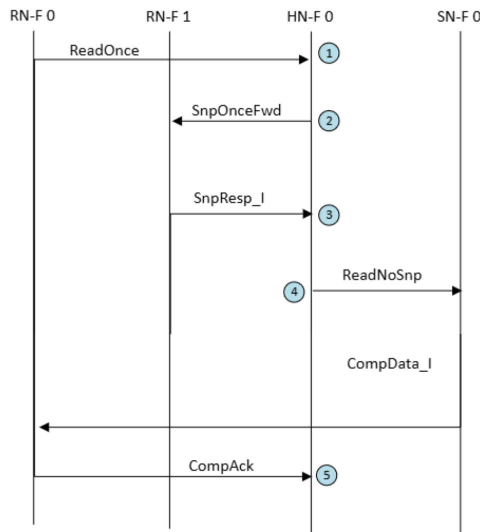

Figure 4. Time-space diagram of a ReadOnce Flow.

In the time-space diagram shown above, the scoreboard starts modelling the next expected outcome after it receives the incoming request at point 1, then matches the expected snoop request when it sees an equivalent transaction from the DUT at point 2 and so on. At all the points marked 1 through 6, the scoreboard keeps recording the relevant packet and any metadata into the coverage object. When the scoreboard sees a CompAck at point 6 which is the last expected transaction for this flow, it goes on to write the life cycle coverage object in to the analysis port which is then consumed by the coverage model.

3.4.2 Coverage Sampling:

The coverage model proposed in this architecture consists of two main parts. The first is an intricate unpacking and sampling logic the processes the incoming life cycle objects and the second is the auto generated per opcode cover group class shown in snippet 3 which is included in the same file, so the sampling logic has access to the appropriate cover group.

Lifecycle coverage object contains certain meta data fields related to which HN is it coming from and other internal status fields like SF & SLC lookup results or other micro architecture specifics like process opcodes for handling the various flows in CHI protocol. Along with design-specific fields, the life cycle object contains separate arrays to hold the RN to HN request packet and the HN to SN request packet. For flows requiring multiple snoops and responses, the respective snoop and response packet arrays hold all the different entries.

The sampling logic is responsible for looking at the opcode inside the request packet and determining which cover group to invoke for the sampling process. In cases where there are multiple snoops and responses the logic will call the appropriate sample functions as necessary to capture all the different types of packets and their various data points.

4. Results

Following the implementation of the Coherent Opcode Flow Coverage Architecture, a phased approach was adopted to integrate and validate it within the broader verification environment. This section, which doubles as the last step in the flow (Regression & Coverage Analysis) outlines how the methodology evolved from initial bring-up to becoming a vital tool for driving stimulus decisions and regression planning.

4.1 Early Bring-Up and Validation

In the initial phase, the focus was on ensuring that the coverage infrastructure, especially the lifecycle sampling logic was functioning reliably. To achieve this, coverage runs were executed using targeted, directed test lists that were handpicked to exercise well-understood opcode flows. These tests served two purposes: First, they allowed for troubleshooting and debugging of early integration issues between the scoreboard, lifecycle object construction, and

coverage model. Second, they established a trust baseline for the coverage data being collected by comparing expected opcode flows with what was sampled and reported. This early bring-up phase proved crucial in validating the end-to-end data path from transaction retirement to coverage sampling, ensuring that only complete and legal transaction lifecycles contributed to coverage metrics.

4.2 Scaling to Weekly Regression Runs

Once the methodology demonstrated correctness and reliability in controlled environments, it was scaled to larger weekly regression runs that included extensive random test lists. These regressions spanned hundreds to thousands of tests and provided a broad sampling of opcode behaviors, snoop combinations, and system states. Integrating the coverage framework into these full-scale regressions marked a significant milestone. It not only maintained robustness under volume but also started to highlight gaps and trends in real time, providing visibility that was otherwise difficult to obtain from log inspection or VIP coverage alone.

4.3 Actionable Insights from Coverage Reports

Almost immediately after deployment in regressions, the coverage framework began producing insightful, actionable data. One of the earliest benefits was the identification of unseen snoop response types. For example, certain snoop transactions (e.g., SnpSharedFwd & SnpCleanFwd) were being issued, but their response combinations (SnpRespDataPtl_I_PD) were never observed in simulation. Additionally, within already-exercised response types, the model exposed missing coverage on specific response and forward state combinations, revealing design paths that had remained untested. These gaps if left unaddressed could have resulted in logic remaining unverified despite high test counts or code coverage. The visibility into which coherence states, data sources (forwarded vs memory), and agent interactions had not yet been exercised proved to be invaluable in directing stimulus development.

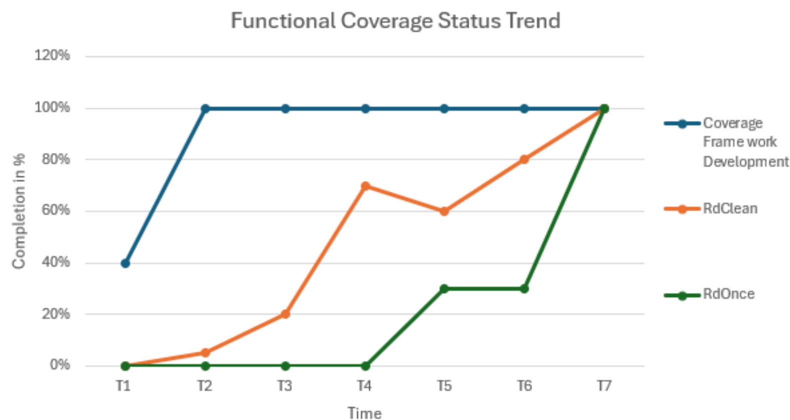

Figure 5. Coverage scores on per opcode flow coverage as reported by the framework over time.

4.4 Feedback Loop for Test and Resource Planning

The coverage data did not remain passive; it actively influenced how test content was developed and prioritized. With clear metrics on what was missing, test writers were able to craft targeted constraints, sequences, or directed tests aimed at closing those specific gaps. This feedback loop between coverage analysis and test creation significantly improved regression efficiency and helped teams focus on “high ROI” stimulus. Furthermore, the data served as a resource planning tool, helping test managers allocate effort based on where the coverage shortfall was, rather than relying solely on intuition or blanket randomization.

4.5 Coverage Stability and Anomaly Detection

Over time, the periodic tracking of functional coverage numbers revealed another benefit: early detection of regressions in coverage quality. In certain cases, drops in coverage metrics—despite test results showing no functional failures prompted deeper investigation. These anomalies led to the discovery of underlying changes in base test layers or common sequences that inadvertently altered the behavior of certain flows, preventing coverage sampling from triggering. Such issues would have gone unnoticed without a dedicated coverage monitoring mechanism and could have silently eroded verification quality.

By surfacing these hidden regressions, the framework proved its value not just in measuring progress, but also in preserving testbench integrity over time.

5. Conclusion

Overall, the integration of the coverage architecture into the regression workflow transformed coverage from a late-stage checklist into a proactive, continuous metric that shaped the project's verification strategy. It enabled deeper observability, faster closure, and smarter use of regression cycles, all of which are critical for first-silicon success in large, configurable, coherent NoC designs.

References

[1] ARM/CHI Specification : IHI0050G_amba_chi_architecture_spec.pdf
[2] A Primer on Memory Consistency and Cache Coherence Second Edition
[3] https://www.openedges.com/