Scalable Formal Verification Framework for NoC System Address Map Configurations

Anishmon Soosai (anishmon.soosai@openedges.com)

Shivaprasad Naranapura Chandrashekara Swamy (shivaprasad.swamy@openedges.com)

Kavin Rajendran (kavin.rajendran@openedges.com)

Kranthi Konganti (kranthi.konganti@openedges.com)

Openedges, Austin, USA

1. ABSTRACT

Modern Network-on-Chip (NoC) architectures increasingly rely on configurable System Address Maps (SAMs) [1] to support the diverse integration needs of multi-core processors, memory subsystems, and specialized accelerators. SAMs define essential system behaviors such as connectivity, routing paths and access permissions. This flexibility enables scalable and modular design across a wide range of applications, from general-purpose computing to domain-specific accelerators. However, it also introduces significant verification challenges. Each SAM configuration represents a unique operational scenario. As design parameters—such as fragmented address spaces, reserved regions, hashing, routing and access policies—vary, the number of possible SAM variants grows rapidly. This explosion in configuration space makes it increasingly difficult to ensure correctness using traditional simulation-based verification methods. Techniques such as UVM randomization, constrained-random verification, scoreboard and coverage-driven regression struggle to explore the full range of configurations within typical project timelines. As a result, subtle corner-case issues—such as address overlaps, misrouted traffic, unintended access, and protocol violations—often go undetected until late in the design cycle, leading to costly debugging and delays. Manually adapting testbenches for each SAM variant is not only labor-intensive and error-prone but also fundamentally unscalable. As NoC designs grow in complexity and configurability, verification teams face increasing pressure to find efficient, repeatable, and automated solutions that can keep pace with design evolution and ensure comprehensive coverage. We developed a scalable formal verification methodology to effectively address this challenge.

2. INTRODUCTION

2.1 Overview of SAM

In a Network-on-Chip (NoC) system, every Requester such as a Request Node (RN) or a Home Node (HN) relies on a System Address Map (SAM) to determine the Target ID (TgtID) [CHI- B13.10.2] of each outgoing request. The SAM defines how different address ranges are distributed across various targets in the system, such as memory controllers, peripherals, or other nodes.

Depending on the design, the SAM can range from a simple static mapping, where all outgoing transactions use a fixed Node ID, to a fully configurable and hierarchical decode that supports multiple address regions and remapping options. This flexibility allows the NoC to adapt to complex topologies involving RN(Coherent/Non-coherent), Subordinate Node(Coherent/Non-coherent), and Home Nodes, especially in cache-coherent architectures where address ownership and routing rules vary by region.

For functional correctness, the SAM must provide a complete and accurate decode of the entire system address space. Any address that does not map to a valid physical component should be redirected to a dedicated error-handling agent capable of generating an appropriate error response. This mechanism ensures graceful handling of invalid access and improves system robustness.

The image below Figure 1 reference design where System Address map Dec(Decoder) block resides in the NOC design. In this configuration, the RN0/Dec logic utilizes the System Address Map (SAM) to decode the incoming address and generate routing identifier: TgtID = HN0..N. Similarly, the HN0/Dec logic applies to the SAM decode in the context of the home agent, producing TgtID = SN0..N(Subordinate Node 0). The output from the Dec block is then used to construct CHI request packets that are transmitted across the network.

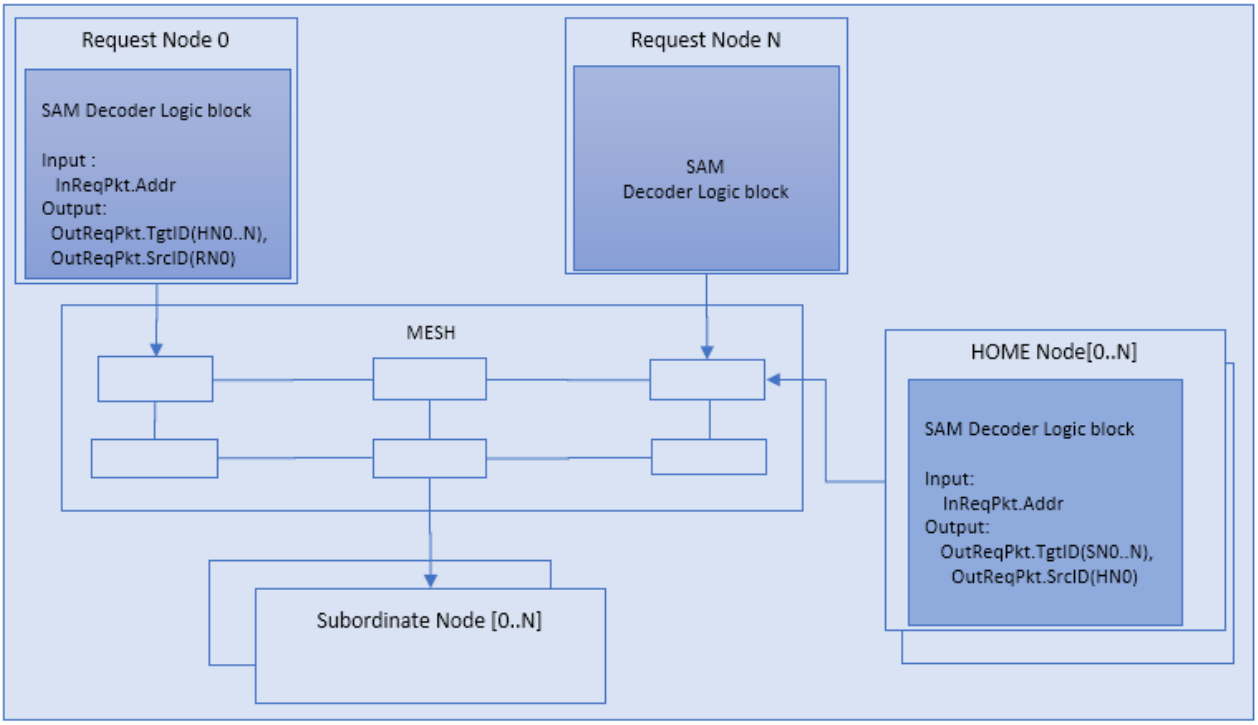

Figure 1. NOC SAM Decoder logic

2.2 Verification challenges

During the initial planning phase, an extensive analysis was performed to define the verification scope and strategy necessary to achieve comprehensive coverage across all System Address Map (SAM) configurations and operational scenarios. This study identified a large number of configuration sets, along with numerous derived variations. Examples include address width variations from 32 to M bits, scalability in the number of home agents from 2 up to N [Max Numbers] with diverse hashing algorithms, and distinctions across secure/non-secure and coherent versus non-coherent agent types as mentioned in Figure 2. Additional complexity arises from configurations involving valid and reserved address regions.

Implementing verification for these scenarios using a traditional UVM-based or block-level environment would have been highly time-consuming, demanding significant effort to build testbenches, test cases, and coverage models. To address this challenge efficiently, we developed a scalable formal verification framework tailored to handle the complexity and breadth of SAM configurations.

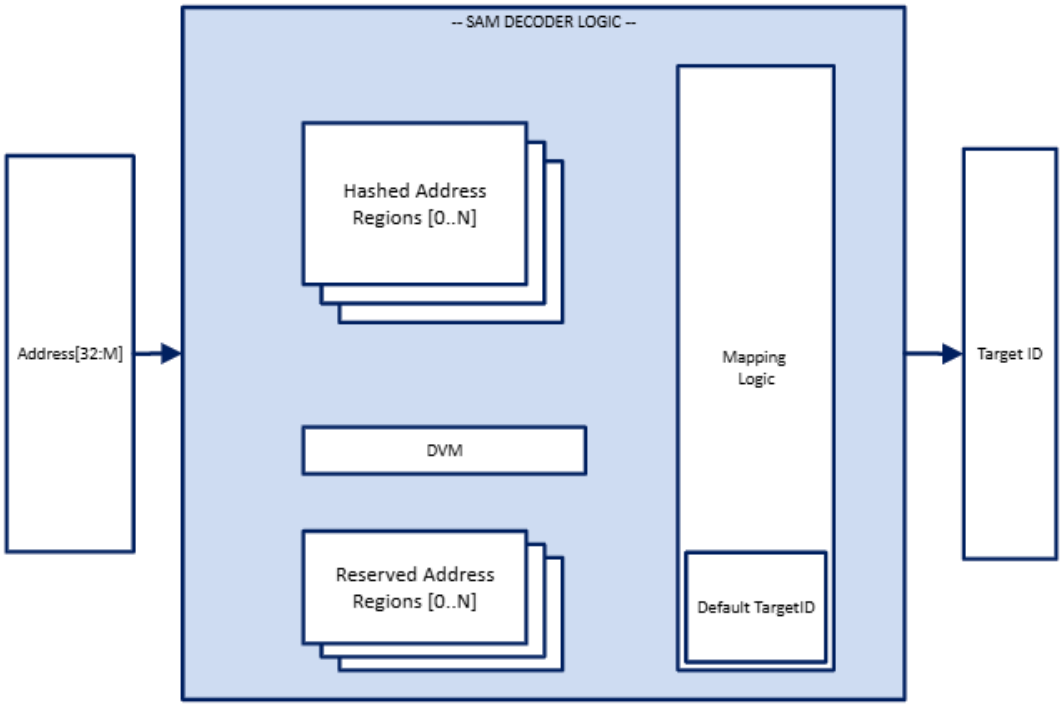

Figure 2. Reference Scalable SAM block

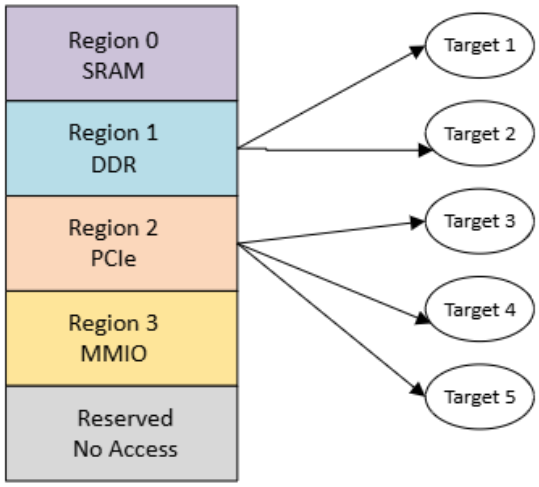

Figure 3. Example Region mapping to multiple targets.

Figure 3 shows an example of an address map with multiple regions. Due to the heterogeneous and non-uniform nature of modern system memory architectures the address map to Target ID decode logic mapping is not always linear. The decode logic must manage distinct address regions, each with unique attributes that dramatically influence the mapping. A particularly critical complication arises when specific high-performance regions utilize address hashing to distribute traffic across multiple end points. This hashing mechanism breaks the straightforward linear relationship between address space and Target ID, requiring formal verification to precisely model the dispersal function to ensure correctness.

3. PROPOSED SOLUTION

The scalable formal verification framework is built to automatically adapt to different design configurations without manual intervention. It intelligently interprets configuration parameters and dynamically generates the required testbench components and assertions based on factors such as the number of regions, hash functions, and related attributes. After generation, the framework runs comprehensive regressions to ensure all properties are successfully

proven. The same steps are then repeated with new configurations until every possible combination in the design space has been covered, delivering complete verification coverage as shown in Figure 4.

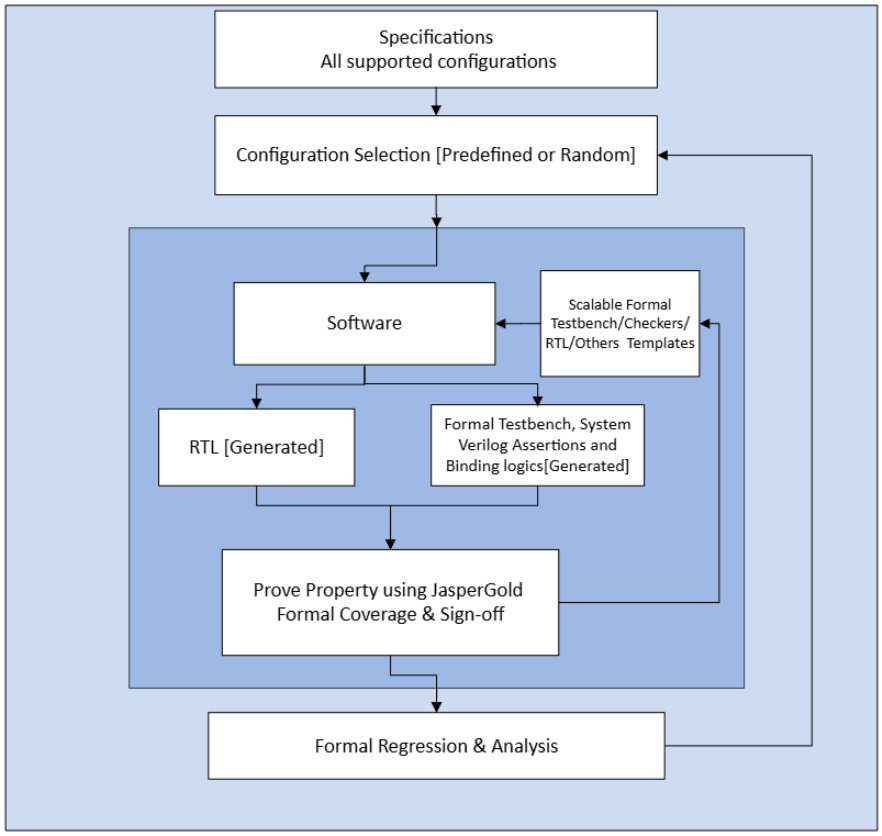

Figure 4. Scalable Formal Verification Framework

The following sections describe Figure 4

3.1 Scalable Formal Verification Framework

3.1.1 Specifications & All supported configurations

The NOC microarchitecture specification outlines the supported SAM configurations, including parameters such as address width, number of home nodes, security attributes (secure/non-secure), valid address regions, and reserved address regions[Refer Figure 3].

3.1.2 Configuration selection

Configuration identification and prioritization are critical to the execution process. These configurations are selected based on sanity validation, design readiness, and stress-test coverage. There are two groups of configuration lists introduced:

Predefined Configuration List: A predefined set of configurations was established to define the execution order according to scheduling requirements.
Examples:
Sanity Testing: Address_Width = 34, 2 hash regions, DVM, and one reserved region.
Maximum Configuration Testing: Address_Width = <Max_Address_Width>, Max_hash regions, DVM, multiple interleaved reserved regions and others.

Randomized Configurations List: Scope of this list is to introduce variations not included in the predefined configuration set.
Example: A reserved region can be placed anywhere within the address map. For instance, some designs position the reserved regions at the bottom, interleaved or top. This process helps to ensure that decoding logic is correctly implemented when address width variations occur across different configurations.

3.2 Software

Software automatically generates RTL, testbenches, runs scripts by reading design configurations and associated settings. The scalable formal testbench is integrated with this software, and both the testbench and checkers are designed to be fully aligned and compatible.

This adapts automatically to varying design configurations by interpreting parameters and dynamically generating components and assertions.

3.3 RTL/Formal Testbench, System Verilog assertion and binding

The formal testbench incorporates prediction logic that models the algorithm and its equivalent decoder functionality to compute expected outputs for verification. RTL output signals are captured as actual values and fed into a checker that performs comparisons against these predicted results as shown in Figure 5. To ensure robust validation, SystemVerilog assertions are implemented for targeted scenarios, verifying that the expected values align with RTL outputs under all operational conditions. Reference demo code included in Figure 6.

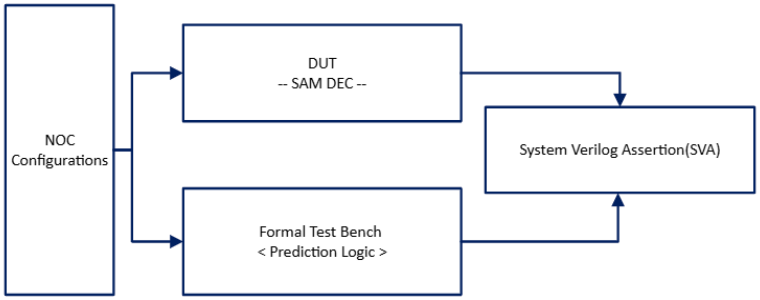

Figure 5. Formal Testbench

3.3.1 Few Key elements from Test plan:

Feature	Test plan
Parameters	Perform static assertion checks for parameter values, such as verifying the size of ADDR_WIDTH, NODE_ID_WIDTH…etc
TgtID, NodeID Check	Model the TgtID and Node IDs in the testbench and ensure they match the actual RTL values
Reverse Lookup logic	Ensure that the node IDs and port IDs returned from the home nodes generate the correct reverse IDs for routing packets to the Request node
Boundary Address	Ensure that address regions are properly aligned and do not overlap with other regions.
Error Scenarios	Verify that reserved regions trigger an error flag and return the default home agent ID as TgtID
Hash Function	Verify Hash equations are getting implemented correctly
…..	….
Assumptions	Review all assumptions with the design team and obtain sign-off
Formal Coverage	Review code coverage, dead code, unreachable code, stimuli, checkers and COI…etc

```
//---------------- Example Demo Code ------------------------------------------
// Inputs from DUT(Decoding logic)
input         dut_valid;
input  [51:0] dut_addr;
input  [15:0] dut_tgtid;

// Generated Code
bit [1:0]  NUM_SAM_REGIONS                          = 3;
bit [`ADDR_WIDTH-1:0] SAM_START_ADDR [NUM_SAM_REGIONS] = '{REGION0_START_ADDR, REGION1_START_ADDR,REGION2_ST_ADDR};
bit [`ADDR_WIDTH-1:0] SAM_END_ADDR   [NUM_SAM_REGIONS] = '{REGION0_END_ADDR,   REGION1_END_ADDR,  REGION2_END_ADDR};
bit [`TGTID_WIDTH-1:0] SAM_TGTID     [NUM_SAM_REGIONS] = '{TGTID0,TGTID1,TGTID2};

//------------------- Common functions -------------------------------
// Example : To Generate TgtID
function bit [`TGTID_WIDTH-1:0]  TbTgtIDGen(input bit [`ADDR_WIDTH-1:0] addr);
  for (int i = 0; i < NUM_SAM_REGIONS; i++) begin
   if (addr inside {[SAM_START_ADDR[i] : SAM_END_ADDR[i]]})
    return SAM_TGTID[i];
  end
  return 'habcd; //  InValid Address region
endfunction

//---------------------  Checkers -------------------------------------
//SVA_01: TgtID compare/Check of DUT Vs Tbench
property sam_tgtid_compare_p;
  @(posedge clk) disable iff (!reset_n)
  dut_valid |-> (dut_tgtid == TbTgtIDGen(dut_addr));
endproperty
SAM_TGTID_VALUE_CHK : assert property (sam_tgtid_compare_p)

//SVA_02: Flag the fail if input dut_addr doesn't find any match
<< ..... >>

//SVA_03: Hash function check
<< ..... >>

//SVA_04: Error flag check
<< ..... >>

//SVA_05: Static/Configuration Checks
<< ..... >>
```

Figure 6: Code snippet of demo checker

Formal testbench is bound at the SAM rtl boundary.

Example binding : bind noc_sam_dec_dut noc_addr_dec_fpv noc_addr_dec_fpv_inst (.*)

3.4 Prove the property using JasperGold and Formal coverage and sign-off

All SystemVerilog assertions were formally proven using JasperGold, Cadence's formal verification platform. During the property proving phase, several design and software bugs were uncovered, including critical issues identified early in the project, which provided significant value to the overall effort. To optimize runtime and reduce complexity, only the SAM logic was retained for formal analysis, while all other design components were black-boxed. This approach enabled efficient property convergence and accelerated bug detection without compromising verification coverage.

The Jasper Design Coverage Verification App was employed for formal coverage analysis to identify dead code, structurally impossible conditions, checker gaps, and code coverage gaps, including toggle, branch, and statement coverage. Extensive reviews were conducted with design teams to ensure proper sign-off of all formal verification activities. The current setup is tool-agnostic and can seamlessly run on alternative formal platforms such as Questa Formal and VC Formal. Additionally, comprehensive manual reviews were performed across multiple configuration variations to validate the correctness of the generation process. During sign-off, all assumption logics were thoroughly reviewed to ensure that the formal flow was not over-constrained. Example Figure 7 from Jasper coverage app gives more details. Black color shows dead code, red color shows unreachable etc.

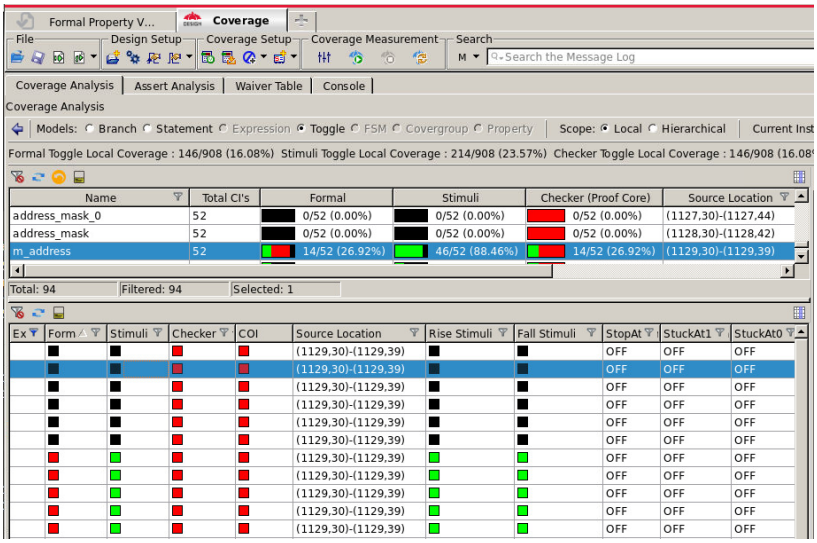

Figure 7: Snippet of coverage details for Formal, Stimuli, Checker

3.5 Formal regression for other configurations

A comprehensive formal regression flow was implemented to systematically execute all configuration combinations defined in the verification plan. After each successful run, the process returned to the configuration selection step to pick new combinations and repeat the execution cycle. This iterative approach continued until every configuration was exercised, all properties were proven without failures, and associated reviews were completed and signed off. The methodology ensured exhaustive coverage across all design variations, minimized gaps in property validation, and delivered a robust verification closure. By automating the regression and integrating review checkpoints, the flow provided predictable convergence, improved efficiency, and reduced the risk of missed scenarios.

4.RESULTS

RTL Findings: Boundary decoding issues were observed between SAM address regions, with formal testbenches revealing out-of-range addresses and overlap related bugs. Reverse lookup logic also showed issues, where certain port and node ID combinations produced incorrect outputs. Additionally, decoding errors for port_id and node_id were found across coherent and non-coherent nodes. Dead code and Unreachable code analysis helped to cleanup redundant code and for adding relevant formal checks.

SW Findings: In simulation-based flows, exercising specific and complex NoC configurations often occurs late in the project schedule. By adopting a formal approach, we achieve rapid turnaround and early validation of NoC design generations, enabling faster detection and resolution of potential issues. Formal verification helped to uncover key user-configurable inputs such as memory size, hash regions and modes, and reserved areas.

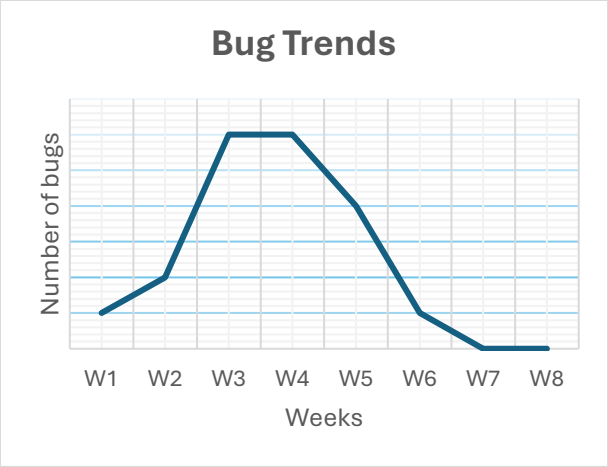

Figure 8. Bug Trends

As illustrated in Figure 8., the bug trend graph shows that the formal verification approach identified numerous RTL and software bugs early in the verification development cycle. The data corresponds to a specific NoC configuration, with similar patterns observed across other configurations. In contrast, traditional simulation typically exposed such issues much later in the verification process.

This methodology is designed to scale efficiently as new SAM variants are introduced. It supports iterative design changes without requiring manual updates to testbenches or assertions. Additionally, it enables parallel formal regression, allowing multiple configurations to be verified simultaneously with minimal overhead. This parallelism significantly improves throughput and reduces turnaround time for verification cycles.

This framework has uncovered multiple NoC/RTL generation issues, including software-related bugs that were difficult to detect using simulation alone. Some of these issues include misconfigured routing paths, incorrect address decoding logic, and unintended access permissions. Formal coverage metrics confirmed that critical edge conditions and complex routing interactions were thoroughly exercised. The framework also identified dead code and gaps in stimuli, enabling engineers to refine their designs and improve overall quality.

Beyond bug detection, the methodology enhances traceability and documentation. It generates structured reports summarizing assertion results, coverage metrics, and configuration-specific insights. These reports help teams track verification progress, identify weak spots, and make informed decisions about design readiness.
By automating repetitive and error-prone tasks, the methodology allows verification engineers to focus on higher-level analysis and architectural decisions. It also promotes reuse and consistency across projects, making it easier to scale verification efforts across multiple designs.

5.CONCLUSION

In summary, our formal verification methodology offers a practical, scalable, and automation-driven solution for verifying highly configurable SAMs in modern NoC designs. By combining formal verification and intelligent automation, we have built a robust framework that meets the demands of today's complex and reconfigurable SoC environments. This approach improves confidence in design correctness, accelerates project schedules, and reduces engineering risk-making it a valuable addition to the verification toolbox for next-generation NoC-based systems.

6. REFERENCES

[1] ARM CHI Specification(Chapter B3): IHI0050G_amba_chi_architecture_spec.pdf
[2] https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html]
[3] Cadence JasperGold reference manual
[4] https://www.openedges.com/
[5] Accelerated, High Quality SoC Memory Map Verification using Formal Techniques, DVCon, 2014
[6] Hierarchical Formal Verification and Progress Checking of Network-On-Chip Design, DVCon, 2025
[7] A Semi-Formal Verification Methodology for Efficient Configuration Coverage of Highly Configurable Digital Designs, DVCon, 2021