Raising the level of Formal Signoff with End-to-End Checking Methodology

Ping Yeung, Arun Khurana, Dhruv Gupta, Ashutosh Prasad, Achin Mittal
Oski Technology
San Jose, CA, Gurgaon India

Abstract - The use of formal verification has been steadily increasing thanks to the widespread adoption of automatic formal, formal applications and assertion-based formal checking. However, to continue finding bugs earlier in the design process, we must advance formal verification beyond focusing on a handful of localized functionalities toward *completely* verifying all block-level design behaviors. An end-to-end formal test bench methodology allows the RTL designer and formal verification engineer to work parallelly to finish design and verification on all functionality formally signed-off as bug-free. Given that today's formal tools cannot close the end-to-end checkers required to verify complex IP blocks, we must rely on methodology to tackle design complexity in a way that allows the formal tool to converge in project time. This paper aims to demystify the end-to-end formal test bench methodology and discusses how we can reduce the complexity of the design with functional decomposition and abstraction techniques.

I. INTRODUCTION

Many companies have used formal verification to verify complex SOCs [1] and safety-critical designs [2]. In addition, formal verification has been used for assurance[2], bug hunting[3], and coverage closure[6][7]. As companies have adopted formal verification, we can classify the usage into five maturity levels, as depicted in Figure 1.

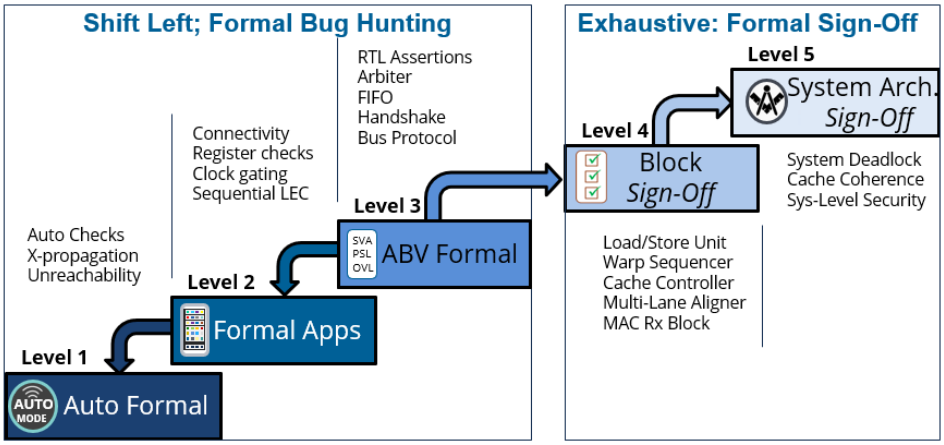

Figure 1: Formal verification usage levels

- Level 1: Automatic Formal such as auto-checking, formal linting, dead-code identification, etc
- Level 2: Formal Applications such as connectivity checking, register checking, X checking, etc
- Level 3: Assertion-based Formal Verification such as interface and bus assertions, embedded assertions, etc
- Level 4: Block-level Signoff that thoroughly verifies all block-level design behaviors.
- Level 5: System Architectural-level Verification focusing on specific high-level design requirements

Level 1 to 3 has been supported well by EDA vendors. Hence, we are not going to cover them. Level 4 introduces *end-to-end checking methodology*, which means the DUT is exhaustively verified, and block-level simulation is completely replaced for that DUT. Once all bugs are found, the DUT is formally signed off with proof that every bug has been found. Level 5 formal aims to prove that the system architecture is correct for specific requirements such as cache coherence or that the system is deadlock-free. System architectural-level verification requires significant expertise, effort, and decomposition of the targeted problem.

This paper focuses on Level 4 block-level signoff as it is a unique reliance usage of formal verification. It provides a good return on investment because you can:

- find more bugs and verify the block while the designer is coding the RTL (early deployment),
- replace simulation entirely and do a formal signoff of the block (exhaustiveness),
- reuse the formal testbench with updated RTL to quickly confirm a fix or find new issues (reusability).

Unfortunately, it is still not commonly used in today's industry because few companies have invested in it, few managers have taken the risk of deploying it, and few engineers know how to use it effectively. This paper tries to address these knowledge gaps by explaining how an end-to-end formal checking methodology can achieve block-level signoff. In addition, we will discuss the formal testbench planning, implementation, and signoff process. Finally, we will explain how to address the design complexity issue with functional decomposition and abstraction techniques.

II. END-TO-END CHECKING METHODOLOGY

The concept of end-to-end formal checking was first introduced in the tutorial of FMCAD [7]. Although it had explained the concept clearly, it did not describe much about the planning and deployment process. To successfully replace block-level simulation with formal signoff, we need to start the end-to-end formal testbench development early in the design cycle. Therefore, commitment from management in the process is essential. Organization-wise, management will acquire formal expertise, allocate formal engineers to partner with RTL designers, and plan for compute resources. Table 1 summarizes the three phases of the end-to-end checking methodology. They are the formal test planning phase, implementation phase, and closure phase.

Task	Planning	Implementation	Closure
Management	Acquire formal expertise	Allocate formal engineer resources	Plan compute and vendor resources
Block	Identify and Evaluate	Capture Interfaces	Validate Constraints
Function	Describe and Prioritize	End-to-End Checkers	Conclusiveness
Complexity	Decompose and Map	Abstraction Techniques	Formal Coverage

Table 1: Formal Planning, Implementation, and Closure of End-to-End Checking Methodology

A. Formal test planning

Formal test planning was first described years ago in [8] and again in [6]. It is the most critical step in the End-to-End checking methodology. The formal test planning process consists of:

- identify the right blocks to apply formal and evaluate the design metrics to determine the effort,
- describe the list of test requirements w.r.t. the design specification and prioritize them,
- decompose complex test requirements into manageable ones and map them to the design

An excellent formal test plan leads the subsequent testbench execution on a path that is predictable and trackable. Thus, it avoids potential problems and delivers the best return on investment for the project. However, the planning processes require past formal signoff experience and intimate design knowledge. Hence, it is best to be done by an experienced formal expert in conjunction with the design team.

Sometimes, the block we want to verify is too complex for formal verification or does not align with the test requirements. In these cases, we will need to take a divide and conquer approach to partition the block explicitly into sub-modules or virtually with SystemVerilog bind statements. Sub-modules can be created with well-defined functionality and interfaces when design teams consider formal complexity early in the design cycle. Then, these sub-modules can be verified cleanly and concurrently. A good example is to have arbitration logic in a submodule. Another one is the IDMA, DMAC, and ODMA submodules in the CNN DMA Controller [4]. On the other hand, System Verilog bind statements can help align the design corresponding to the test requirements. For example, a bind statement can be used for each interface of the design. Furthermore, additional bind statements can focus on the different data flow of the design and data integrity at the data transport parts of the design. Using these bind statements, we have a powerful way to partition the design corresponding to the test requirements not limited by the design hierarchy.

Test requirement mapping is another common problem. For example, the design specification or the test requirement may discuss some high-level behaviors, such as data starving, back-pressure, or head-of-line blocking. As these concepts and conditions are not generally captured explicitly in the RTL code. Extra monitoring models will need to be written to detect these concerned cases.

B. Formal Testbench Implementation

It is the implementation phase of the formal test plan and requires careful partnering of formal engineers with design team members. Verification tasks should be assigned and tracked at this phase. It involves:

- capturing the interface properties as assertions or constraints,
- implementing end-to-end checkers to verify the behavior of the design and,
- planning and preparing abstraction models when necessary.

Interface properties used as output assertions for one block will be used as input constraints for another. Therefore, it is helpful to make sure they can be switched easily. End-to-end checkers serve as a reference model of the design-under-test (DUT). When we engage with design teams early in the process, the end-to-end checker is developed side-by-side with the RTL code. It references the same design specification as the RTL design. Hence, it can provide early feedback on the correctness of the design, ensure its quality, and evolve with the design changes. Abstraction models are techniques to help formal verification short-circuit the depth and reduce the state space of a design. They will not be needed until the formal testbench environment is up and running. However, it will be late if some properties fail to converge and abstraction models are needed late in the schedule. In addition, some complex abstraction models will take time to develop. Therefore, it is much better to plan for them, especially for data integrity-related abstraction models. To smoothly execute the formal testbench implementation phase, skillsets such as writing efficient formal test benches, understanding essential formal tool features, capabilities, and limitations, and applying formal techniques to manage design complexity are crucial.

Similar to other verification processes, a formal testbench implementation process is very iterative. For example, it is common for an end-to-end checker to fail in shadow depth initially. However, during the debugging process, it enables a deeper understanding of the design, leading to a refinement of the checker and assumptions. This refinement process ensures the formal tool explores all corner-case scenarios from easy to complex until the end-to-end checker is validated.

C. Formal Closure

This is the final closure phase that determines if the formal testbench has reached signoff status. To ensure the formal testbench has reached signoff quality, one must

- validate the constraints to make sure they are not over-constraining,
- prove, debug, fix and conclude all the end-to-end checkers,
- examine and determine that the formal coverage is sufficient

However, before looking at the qualitative measurements, one critical question (to ask one final time) is: have we verified all the dead-on-arrival, worrying, and corner cases. Most design specifications describe the expected behavior of the design; seldom will they talk about the error and unexpected scenarios. One job of the formal engineers is to think of the potentially distressing situations, ask questions, capture them and verify that they will or will not happen. A practical approach is to examine the bugs found by other verification approaches such as simulation and emulation. Although the bugs may have already been fixed, have they been verified thoroughly against all scenarios? For a known bug, formal verification is good at finding all possible triggers and ensuring that the fix will cover all possible scenarios.

The most practical way to verify constraints and assumptions is to integrate them into the simulation regression. We want to ensure there is no unintentional over-constraint. However, observing simulation coverage of these properties is essential. We need to make sure they are being exercised by simulation. Similar to simulation coverage, formal coverage can be measured using line, branch, state, and transition coverages. Cone-of-influence (COI) coverage can be used early in the implementation phase to ensure sufficient end-to-end checkers cover the whole design. Then, at the closure phase, formal core coverage can measure how much formal verification has exercised the design. To ensure the design specifications are verified, end-to-end checkers should be proven or have reached the Required Proof Depth (RPD)[9]. The RPD depends on the micro-architecture of the design. Therefore, it is calculated using several

factors, including the latency of the design, the reached interesting corner-cases, the fired counter-examples, etc. Finally, there may still be formal failures (counter-examples) that are exceptional cases. We can document them literally as unsupported scenarios. However, the best way to document them is to specify the constraints or assumptions required to make them conclusive.

{TBD} based on our experience, project teams always under-estimate the computational resources and tool vendor support to perform formal coverage and closure at the end of the project.

III. CREATION OF END-TO-END CHECKERS

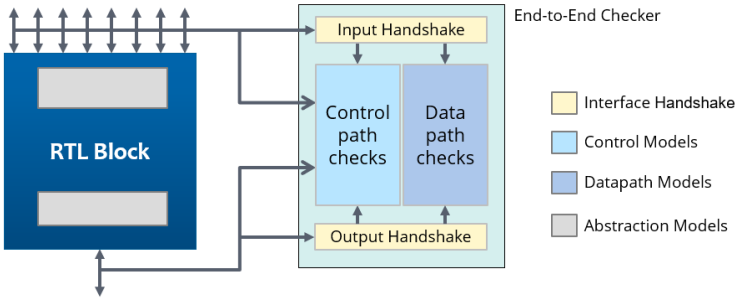

Figure 2: The structure of an end-to-end checker

In order to sign off a block with formal verification, the Level 3 formal bug hunt approach is insufficient. The locally embedded assertions rely on the correctness of the RTL code to catch corner-case bugs. For instance, a FIFO will need to store data before it overflows. Hence, if formal verification is to be relied upon as a primary verification methodology for a design, end-to-end checkers must be used to capture the functionality of the design. When compared to low-level assertions, end-to-end checkers are much more comprehensive as they:

- Provide clear value to the project team because they map directly to the functional specification,
- Identify incomplete or ambiguous specifications early in the design cycle,
- Provide the highest return on investment as it can identify deep or unaware corner case issues.

An end-to-end checker uses RTL code to describe a reference model for the required behavior of the block. We roughly divide it into four components. As shown in Figure 2, it consists of input handshake, output handshake, control path, and data path. For each interface of the design block, the input handshake captures the interface properties and uses them as constraints. In addition, it extracts the control and data signals for the reference model. The output handshake makes sure that the output signals follow the defined protocols. It also extracts the control and data signals to check their values by the reference model. Not surprisingly, proving the end-to-end checkers is usually computationally much more complex than local checkers. It is one of the reasons that we recommend separating the control and the datapath logic. It allows us to focus on the control logic first. As we will describe in the next section, various abstraction techniques can be used differently on the control and the datapath logic. The control logic is normally modeled as finite state machines in the RTL code to be efficient and compact. However, complex FSMs with many input signals and fan-in logic may become bottlenecks for formal verification. Since the formal testbench is not going to be synthesized, we can code the control logic as multiple FSMs, or sequences that can be enabled based on the operational functionality of the design.

Finally, to give some examples, in the multi-cast crossbar design [6] and the CNN DMA controller design [4], they consist of the following end-to-end checkers:

- Control path end-to-end checkers:
  - An arbitration checker (a combination of two checkers) for the arbitration scheme
  - A consistency checker to ensure no spurious grant is given to a client
  - Predict and compare checkers for address correctness of the DMA load/store operations
  - Performance checkers to ensure read/store operations are performed in each cycle when the conditions are met.
- Data path end-to-end checkers:
  - A data integrity checker to ensure correct transfer from the client to the desired target i.e. data is not corrupted, duplicated, reordered, or dropped.

- Data integrity checkers to ensure correct transfer 1) from read data input port to buffer, and 2) from buffer to store output port.

To verify data transportation logic, we use the Wolper coloring technique [6]. It is a practical approach that doesn't require any data stored in the checker logic for data integrity checking. When the input is constrained to follow the Wolper coloring sequence, as in figure 4, we verify if the same sequence is received at the output. Once set up, formal verification will find ways to break the sequence. However, if proved, it confirms that the data is correctly transferred; it is not corrupted, duplicated, reordered, or dropped.

The rules for generating or verifying Wolper sequence are:

1. If the first 1 is seen, next input/output should be 1
   wolper_1st_1_seen_next_1: (first_one && !second_one && input_valid) |-> (colored_input == 1'b1)
2. If two 1's are seen, only 0's should be seen
   wolper_2nd_1_seen_forever_0: (second_one && input_valid) |-> (colored_input == 1'b0)

Figure 4: The rules of the Wolper sequence

IV. Adoption of Abstraction Techniques

If we want to verify a block's end-to-end behavior, design complexity is one of the success factors [3] we need to conquer. Although tool vendors have been working hard to improve the tool capability, adopting efficient abstraction techniques is still the most efficient way to help formal verification deliver conclusive results and coverage closure. The process includes

- Identify potential areas of design complexity
- Formulate abstraction techniques
  - Applying design domain knowledge to determine suitable abstractions
  - Leverage assume/guarantee reasoning and symmetries in the design
  - Reuse abstraction models for common elements such as FIFOs, memories, etc
- Deploy abstraction models and evaluate the efficacy

Formal verification tools have been advanced sufficiently to identify potential areas of design complexity. By examining the core-of-influence (COI), we can find the design elements that have caused the formal engines to stall. A set of commonly used abstraction techniques and models are summarized in Table 2. These abstraction techniques and manually crafted abstraction models can reduce the required proof depths and the state space of a design to make formal runs less computationally expensive.

Abstraction Technique	Design Complexity	Formal Efficiency
Case splitting	Multiple runs with different cases reducing design complexity per run/case	Reduce COI, reduce state space per run/case
Cut-point/ Black box	Eliminate logic before cut-points/blackbox; add constraints	Increase flexibility but controlled with constraints
Reset abstraction	n.a.	Reduce access depth
Counter abstraction	n.a.	Reduce the length of counting
Symmetric data elements [7]	Eliminate multiple dimensional data elements; add single dimension abstraction model	Reduce COI with symmetry
FIFO [7]	Eliminate logic before cut-points; add abstraction model	Reduce the depth of the FIFO
Data independence [7][6]	Eliminate all storage elements; add Wolper FSMs	Reduce COI with pattern
Memory abstraction [7]	Represent one location instead of the full size of the memory	Reduce COI with symmetry
Tagging [9]	Represent one tag instead of the complete linked list	Reduce COI

Table 2: Abstraction Techniques and Models

Case splitting decomposes a complex design into multiple formal runs. Each run is constrained to focus on a particular configuration or case. As a result, it reduces the cone-of-influence (COI) and the state space of each formal run. Abstraction using cut-points and black-box are similar. Both approaches remove the complex logic from the COI. Thus, these cut-points and the outputs of the black box become *control points* for formal verification. We can add appropriate constraints. The goal is to reduce complexity and increase flexibility for formal verification. Resettable registers and counters are common elements in a design. However, it may take a long sequence of activities to load an interesting value into a critical register or counter. Reset and counter abstraction short-circuit the process and enable formal to control these elements more directly. These approaches have been supported by various tool vendors effectively.

On the other hand, more advanced abstraction techniques will require a good understanding of the design and modeling. For example, for data transport designs such as memory controllers, cache controllers, or DMA controllers, there are a lot of symmetric data elements in the design. To reduce complexity, we can write checkers to verify one symbolic address of the data element only. Thus, it will reduce complexity from SIZE*WIDTH for all data elements to 1*WIDTH for the symbolic address. Table 3 below shows how an abstraction model is different from the original RTL. When RTL reads the memory location corresponding to the symbolic address, the value last written is returned. At the same time, end-to-end checkers will check data integrity corresponding to the symbolic address only.

RTL model	Abstraction model

element_type [SIZE-1:0] element;
 	
element [addr] = wr_data;
rd_data = element [addr];
	element_type abs_element;
	
	if (addr == sym_addr) abs_element = wr_data;
	if (addr == sym_addr) ? rd_data = abs_element;

Table 3: The abstraction model for symmetric data elements

Deep FIFOs are challenging for formal verification in general. An easy way to reduce the design complexity of a FIFO is to minimize the DEPTH parameter. However, this approach cuts the verification state-space forcefully. A more mature approach is to enable formal verification to explore and set the FIFO depth as needed. As illustrated in Figure 5, a symbolic depth is defined and used in place of the DEPTH parameter. The assume property ensures that the sym_depth, once set by formal, remains constant throughout the process. The sym-depth can be as small as 2 or a more significant value if the number of entities is essential for formal coverage. Then, the FIFO can be filled quickly, and the subsequent back-pressure logic can be verified.

```
wire [$clog2(DEPTH)-1:0]  sym_depth;
assume property:          (sym_depth > 1 && sym_depth < DEPTH) ##1 $stable(sym_depth)
abstraction model:        replace DEPTH with sym_depth
assert property:          pop |-> (data_out == mem[rd_ptr])
```

Figure 5: The abstraction model of a FIFO

When data is transported in sequence without modification, we can use the Wolper coloring technique [6], illustrated earlier in Figure 4. It verifies the data integrity of the RTL structure. Furthermore, this approach can be combined with the other abstraction technique, such as the symbolic depth, to verify the end-to-end integrity of the FIFO and its memory. On the other hand, when data is stored and used later, we can represent the memory as one symbolic address instead of the complete array. It leverages the same approach as the abstraction model for symmetric data elements in Table 3. As illustrated in Figure 6, there is a symbolic address and one memory storage. Data is stored and retrieved from the single memory location when the address matches the symbolic address. With formal verification, it will explore all possible values for the symbolic address.

```
RTL memory:               reg [WIDTH-1:0] mem [DEPTH-1:0];
abstraction memory:       reg [WIDTH-1:0] mem;
assume property:          (sym_addr < DEPTH) ##1 $stable(sym_addr)
abstraction write:        if (wr && (wr_addr == sym_addr)) mem <= wr_data;
abstraction read:         if (rd && (rd_addr == sym_addr)) rd_data = mem;
```

Figure 6: The abstraction model of a memory

In design with a scheduler or allocator, incoming requests are stored as tags. As tags can be created and freed out of order, they are managed as a linked list. Therefore, instead of handling a tag manager with 1000s of tags, we replace it with an abstraction model that contains only one symbolic tag. As shown in Figure 7, the assume property is used to ensure the symbolic tag value remains constant throughout the formal run. Then, a simple state machine, with IDLE and STAG states [9], is used to represent the allocation and freeing of the tag. In addition, assume and assert properties are added to the output and input ports of the tag manager:

- Assume property: ensure when the sym_tag has been stored, it cannot be granted again.
- Assert property: ensure when the sym_tag has been freed, it cannot be returned again.

```
assume property:          ##1 $stable(sym_tag)

IDLE → STAG transition:   grant && (grant_tag == sym_tag) – stored sym_tag
STAG → IDLE transition:   return && (return_tag == sym_tag) – freed sym_tag

output assume property:   (state == STAG) |-> (!empty)
output assume property:   (state == STAG && grant) |-> (grant_tag != sym_tag)

input assert property:    (state == IDLE && return) |-> (return_tag != sym_tag)
input assert property:    (state == IDLE) |-> "sym_tag is eventually returned"
```

Figure 7: The abstraction model of a tag

V. RESULTS

Many companies have successfully deployed block-level formal verification signoff with the end-to-end checking methodology. Besides finding bugs early in the design cycle, they have also experienced additional benefits. For example, with the end-to-end checking formal testbench already in place, some projects added late-stage features and confidently debugged them. More importantly, no bug was found in the blocks afterward in integrated simulation regression, system-level simulation, and the designs were taped out successfully.

A. Shader Sequencer of a GPU design, AMD [5]

The shader core is a Single-Instruction Multiple-Data (SIMD) machine with parallel processing across hundreds of execution units. Thus, the Sequencer must fetch, decode, resolve dependencies, issue instructions to multiple execution units, permutations of asynchronous events (traps, exceptions, etc.), mode changes, and control flow scenarios (branch, jump, etc.). The complete end-to-end checking methodology for this block is summarized in Table 3. 17 bugs were found during the process, and some were considered impossible to find in a simulation environment.

Task	Planning	Implementation	Closure
Management	Formal expert (10+ yr) (20% time) Schedule: 3-4 months	Formal engineer (5+ yr) Formal engineer (1+ yr)	16-core, 256GB server
Block	Divide and conquer: Submodules: Block_A, Block_B, Arbiter	Capture Interfaces: External interfaces A2B, B2A checkers	Validate Constraints: Simulation integrated
Function	Prioritize: Data correctness Arbitration workload Sequences and dispatches	End-to-End Checkers: 117 Channel arbitration Tagging Data integrity Forward progress	RTL Bugs: 34 17 critical bugs / Shift-left verification schedule for new features

Complexity	Decompose: Requests (valid, utilization, priority, tag) Data (valid, integrity, progress)	Abstraction Techniques: Retry timer	Formal Coverage: Line: 99% COI: 92% Formal Core: 82%

Table 3: Summary of E2E Checking Methodology for Shader Sequencer of an GPU design

B. CNN DMA Controller of an Embedded Vision (EV) Processor [4]

The DesignWare Embedded Vision (EV) Processor contains a Convolutional Neural Network (CNN) engine for object detection and classification. A key component of CNN is the DMA controller unit. We have summarized the tasks and results of the end-to-end checking methodology in Table 4 below. The DMA has three primary subcomponents. The DMA Controller (DMAC) buffers DMA requests and issues them to the Inbound DMA (IDMA) and Outbound DMA (ODMA) blocks, which respectively control transfers to and from local memory. Formal verification with end-to-end checkers was performed after simulation regression had been completed. However, 6 corner-case bugs were found. In these cases, multiple preconditions had to occur with specific timing to reach the states that would trigger these bugs. Hence, these bugs were highly resistant to simulation regression. We also confirmed that formal verification had complete control of the design code coverage points with the help of abstractions.

Task	Planning	Implementation	Closure
Management	Formal expert (20% time) Schedule: 4 months	Formal engineer (3+ yr) Formal engineer (2+ yr)	16-core, 256GB server
Block	Divide and conquer: Submodules: IDMA, DMAC, ODMA, FIFOs	Capture Interfaces: External interfaces Internal submodule I/Fs	Validate Constraints: Simulation integrated; cross-proved
Function	Prioritize: All IP block, all checks are important	End-to-End Checkers: Data integrity Address correctness Performance	Formal after simulation regression 6 corner-case bugs
Complexity	Decompose: IDMA (read, store, finish)	Abstraction Techniques: Copy Volume Abstraction Address Abstraction (De)compression	Formal Coverage: Line: 100% Condition: 100% COI: 96%

Table 4: Summary of E2E Checking Methodology for CNN DMA Controller of an EV Processor

C. Network-on-Chip (NOC) Configurable Cache Controller

The cache controller is one kind of design that we have verified numerous times in the past years. A recent one is a configurable cache controller in a complex NOC design. There are many configurations and modes of operation that simulation alone would be insufficient. The complete end-to-end checking methodology is summarized in Table 5 below. To handle the complexity of this design, we developed two separated formal testbenches:

1. Tag flow: smaller latency, abstraction memories, operates on all ways in the set
2. Data flow: long latency, abstraction memories, focuses on transaction failures in some

In addition, we maintained a clean and well-understood interface between these two testbenches. The properties on the tag/data flow interface were cross-proved to ensure completeness.

Task	Planning	Implementation	Closure
Manage-ment	Formal expert (10+ yr, 25% time) Schedule: 5-6 months	Sr. Formal engineer (50% time) 2x Formal engineer (2+ yr)	16-core, 64GB server 16-core, 512GB server
Block	Divide and conquer: Submodules: arbiters, cacheline controller, DDR controller, FIFOs	Capture Interfaces: Cmd interface Register interface Data SRAM interface DDR RAM interface Tag <> data interface	Validate Constraints: Simulation integrated; cross-proved
Function	Prioritize: All LRU arbiter (module) Cacheline (SV bind) Tag flow path (SV bind) Data flow path (SV bind) 4x interfaces (SV bind)	End-to-End Checkers: Tag flow: - Tag state - Eviction address/state - Replacement policy Data flow: - Write/read data integrity - Eviction data	Total 496 properties 76% proven 24% bounded 76 bugs 29 bugs are simula-tion resistant
Complexity	Decompose: Tag and Data flow paths were decomposed	Abstraction Techniques: Reset abstractions Cut-points Symbolic sets for symmetric data in tag and data memories Data coloring for data con-sistency	Formal Coverage: Functional coverage Assertion precondi-tion coverage Checkers reach re-quired proof depth

Table 5: Summary of E2E Checking Methodology for NOC Configurable Cache Controller

VI. SUMMARY

We have introduced the end-to-end checking methodology to raise the level of formal verification to perform block-level formal signoff. We have summarized the creation of an end-to-end formal testbench. It is a 3-step process consisting of formal testbench planning, implementation, and closure. To help project teams understand the work better, we covered two essential tasks in-depth: end-to-end checkers and abstraction techniques. For end-to-end checkers, we explained their benefits and their components. For abstraction techniques, we summarized some techniques supported by formal tools and some advanced modeling approaches. Finally, we illustrated the end-to-end checking methodology with three examples detailing the tasks and the results in the 3-step process. From these examples, managers can see that for an SoC design with approximately 50 blocks, if the project plans to deploy formal signoff on a quarter of the blocks, it will require 3 to 4 formal experts plus 20 or more junior to senior formal engineers.

VII. ACKNOWLEDGMENT

We want to express our gratitude for the support of Vigyan Singhal, Craig Shirley, and the whole Oski Team in Gurgaon, India.

VIII. REFERENCE

[1] Ram Narayan, “The future of formal model checking is NOW!”, DVCon 2014.
[2] Mandar Munishwar, Vigyan Singhal, et al., “Architectural Formal Verification of System-Level Deadlocks,” DVCon 2018.
[3] Ping Yeung, et al., “Formal Verification Experiences: Spiral Refinement Methodology for Silicon Bug Hunt,” DVCon 2021
[4] Bruno Lavigueur, Roger Sabbagh, et al., “Finding the Last Bug in a CNN DMA Unit,” DVCon 2020
[5] Vaibhav Tendulkar, Chirag Dhruv, “Formal Verification of a GPU Shader Sequencer,” DAC 2019
[6] Ipshita Tripathi, Ankit Saxdna, et al., “Process & Proof for Formal Signoff - Live Case Study,” DVCon 2016
[7] Prashant Aggarwal, et al., “End-to-End Formal using Abstractions to Maximize Coverage,” FMCAD 11
[8] Harry Foster, et al., “Guidelines For Creating a Formal Verification Testplan,” DVCon 2006
[9] NamDo Kim, Junhyuk Park, et al., “Signoff with Bounded Formal Verification Proofs,” DVCon 2014