

Formal Verification in the Age of Generative and Agentic AI

Debapriya Chakraborty

Debapriya.Chakraborty@asteralabs.com

Astera Labs, Helios Business Park, Wing-D, Ground Floor, Kadubeesanahalli, Bengaluru, Karnataka 560103

Pallavi Atha

Pallavi.Atha@asteralabs.com

ABSTRACT

The exponential rise in SoC complexity, driven by heterogeneous compute, AI acceleration, and advanced power and security architectures, has rendered Formal Verification no longer a choice but a critical necessity. It is uniquely capable of proving mathematical correctness across all reachable design states hence providing exhaustive coverage guarantees. However, scaling FV is challenging, as it demands significant expertise to author SystemVerilog Assertions (SVAs) or behavioral reference models. Generative AI, Retrieval Context, and Agentic AI help broaden FV adoption, accelerating FV-Plan creation, SVA coding, constraint debugging, failure analysis, and workflow automation. Case studies on simple to medium-complexity design blocks show that the proposed AI-assisted methodology achieves at least 70% effort reduction in the FV lifecycle. Standalone LLMs improved productivity but showed requirement gaps and hallucinations. Retrieval context improved coverage and accuracy, while Agentic AI orchestrated proof execution and closure with minimal human intervention, showing clear progression in productivity and verification quality.

1. Introduction

Traditional simulation, being a key methodology in hardware verification, checks design behavior effectively within the scope of authored test scenarios; beyond that scope, however, lies a vast unverified state space that can potentially harbor corner-case bugs. Formal Verification (FV) provides a complementary approach by mathematically analyzing the behavior of a design against a set of formally expressed requirements. Therefore, the importance of exhaustive nature of formal verification practices has grown hand in hand with the evolving architectural complexity of modern SoCs.

Now the challenge is to scale FV as it requires substantial expertise, time, and deep familiarity with both the specification and the design to accurately author SystemVerilog Assertions (SVAs) or a complete behavioral reference model. Engineering productivity involving right translation of complex specification requirements into formal properties, identification

of corner cases, iterative debugging of assertion failures until proof closure naturally gets outpaced by the present-day intricacy of the hardware designs.

Breakthroughs in Artificial Intelligence (AI), including Generative AI, Retrieval-Augmented AI, and Agentic AI systems, help address this potential challenge of broader adoption of formal verification. Generative AI or LLMs can assist FV engineers in transforming natural-language requirements into candidate verification artifacts, generating SVAs, developing reference models, and interpreting formal results. The flow of events occurs based on the interactions through human prompts. Retrieval-based approaches can augment the model with design-driven and project-specific information, thereby improving contextual relevance. Agentic AI extends this capability by enabling an AI system to orchestrate multiple stages of the verification workflow, including artifact generation, tool execution, failure analysis, and iterative refinement.

However, the application of AI to formal verification introduces a different class of challenges. A generated assertion may compile successfully but still be semantically incorrect. An AI system may omit a requirement, introduce behavior not supported by the specification, generate inappropriate assumptions, or fail to identify important corner cases. Therefore, evaluating AI-assisted formal verification requires metrics that capture not only productivity but also verification quality which is assessed in the observation section of this paper.

This work presents an AI-assisted FV methodology evaluated across manual, LLM-based, retrieval-augmented, and Agentic AI workflows on representative design blocks of simple to moderate complexity. Our experiments are primarily conducted on JasperGold FV tool from Cadence and are also compatible with VC Formal from Synopsys and open source tool Yosys.

2. Formal Verification Flow and its challenges

2.1. Conventional Formal Verification Lifecycle

A typical formal verification flow can be divided into the following major stages:

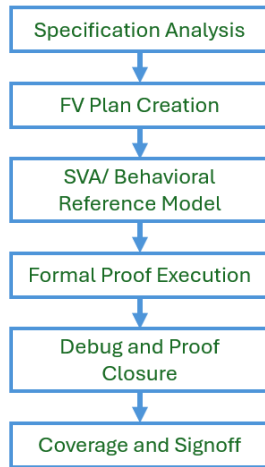


Figure 1: Conventional Formal Verification Flow

2.1.1 Specification Analysis: The FV engineer studies the design specification to extract verifiable requirements, identifying functional behaviors, protocol rules, boundary conditions, and corner cases that must be formally proven.

2.1.2 FV Plan Creation: A comprehensive Formal Verification plan is authored, mapping specification requirements to one or more formal properties, defining the scope, property types, and proof strategies.

2.1.3 SVA/ Behavioral Reference Model: SystemVerilog Assertions (SVAs) are coded to capture formal properties. In some cases, a complete behavioral reference model including necessary constraints is developed to serve as a golden specification for correctness comparison against the RTL design.

2.1.4 Formal Proof Execution: The formal verification tool is invoked against the design, mathematically exploring all reachable design states to either prove the design correctness or show up a counterexample demonstrating a violation of expected functionality.

2.1.5 Debug and Proof Closure: Assertion failures, vacuity issues from over-constrained assumptions, and unreachable cover points are systematically diagnosed and resolved through iterative debugging until all properties are either infinitely proven or reach a safe bound which gets analyzed before the final signoff described next.

2.1.6 Coverage and Signoff: It is the final stage of the Formal Verification flow, at which there are no vacuous passes, all cover points are reachable, no assertion fails, and all bounded proofs have been carefully analyzed for meeting the calculated safe depth. The verification team must have sufficient confidence in the correctness of the design, as qualified by the signoff checklist review, while ensuring more than 99% coverage to formally close the FV effort.

2.2 Automation for FV tool invocation to generating proof reports

Translating the FV plan and authored SVAs into actual proof results requires a reliable and repeatable execution infrastructure. To this end, an in-house make runfv automation flow is used, providing a unified interface that handles filelist expansion, environment setup, TCL generation, FV tool invocation, and results collection in a single streamlined pipeline (Figure 2). The following gives an overview of the automation steps:

2.2.1 Makefile dispatches to runfv.py: The Makefile resolves the block name, FV flow type, tool backend, and run directory, then passes these as arguments to a shared runfv.py script.

2.2.2 Filelist Expansion (.fc to .gf): runfv.py calls expand_vf.py to process the filelist, resolving environment variables, expanding include files, and emitting a flat .gf filelist that the formal tool can directly consume.

2.2.3 FV Rundir Setup: runfv.py creates or verifies the FV run directory structure, including log and output directories, and checks whether a TCL script already exists.

2.2.4 TCL Generation (First Run Only): If no TCL script exists, runfv.py auto-generates a scaffold containing ordered FV Tool commands including design compilation, clock/reset setup, constraint activation, and report generation. On subsequent runs, the existing TCL is reused.

2.2.5 FV Tool Invocation: runfv.py launches FV Tool (from any EDA Vendor) in batch mode. The tool sequentially compiles the design, elaborates the cone of logic, resets simulation, activates constraints, and runs multiple proof engines simultaneously.

2.2.6 Output Artifacts: FV Tool writes results to structured output files including a summary report, a detailed per-property report, and an internal database with counterexample traces for waveform debugging.

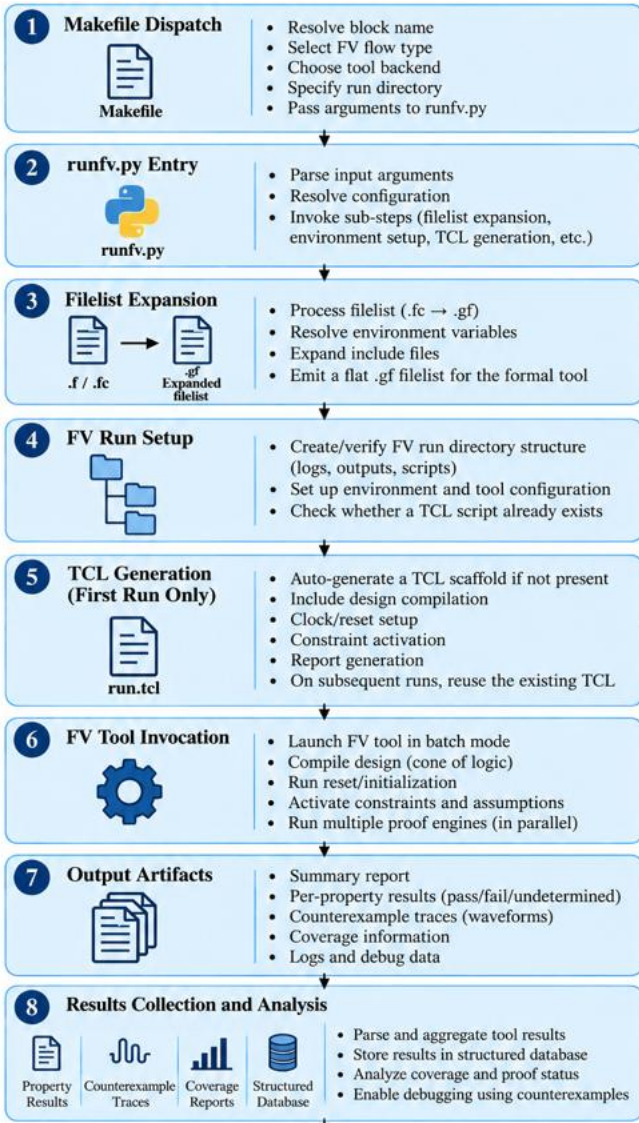


Figure 2: make runfv flow for automation of FV tool invocation to generating proof reports.

2.3 Crucial FV Productivity Bottlenecks

Three challenges are particularly relevant to scaling formal verification.

2.3.1 Requirement-to-Property Translation: Natural-language requirements frequently contain implicit temporal relationships, conditions, exceptions, and dependencies. Translating such requirements into precise temporal properties requires substantial engineering expertise. For example, a requirement describing ordering between requests and responses may need to account for multiple outstanding transactions, simultaneous requests, response ordering, reset behavior, cancellation conditions, backpressure, and legal versus illegal transaction sequences. An assertion that captures only the nominal sequence may compile and prove while failing to represent important aspects of the requirement.

2.3.2 Corner-Case Identification: Formal verification is particularly effective at exposing corner cases, but those cases must first be represented by appropriate properties and environmental assumptions. Missing corner cases in the specification-to-property translation can therefore result in verification gaps even when the formal engine successfully proves all supplied properties.

2.3.3 Iterative Debugging: Formal failures require interpretation of counterexamples and determination of their root cause. The engineer may need to examine RTL behavior, property semantics, assumptions, protocol requirements, signal relationships, state transitions, and formal tool traces. This iterative process can significantly increase the time required to reach proof closure.

3. AI-Assisted FV Methods

Recent progress in AI presents a compelling opportunity to intelligently augment the formal verification lifecycle. Rather than replacing the formal engine, AI serves as an assistive layer. The proposed methodology evolves through three levels of AI assistance: (1) LLM-based generation, (2) LLM with retrieval context, and (3) Agentic AI-based formal verification. (Figure 3)



Figure 3: Types of AI-assisted Formal Verification Methods

3.1 LLM-Based Formal Verification

The first level uses an LLM to assist the formal verification engineer in generating verification artifacts. Typical inputs include human prompts about signal descriptions, protocol behavior, RTL snippets of typedefs, sequence of events, data storage model, latency awareness etc. Back and forth conversation helps LLMs to successfully generate candidate artifacts such as SystemVerilog Assertions, assumptions, cover properties, reference-model code and FV-plan entries. Additionally, when guided with adequate timing data from failure waveform, it can also root cause counterexamples and suggest probable fixes. We have used Claude Sonnet 4.6 AI Model on Open WebUI platform.

3.2 Retrieval-Augmented Formal Verification

The second level introduces retrieval-based context to address one of the fundamental limitations of generic LLM generation:

insufficient knowledge of the specific design and verification environment. A retrieval system can provide relevant information in the form of resources such as design microarchitectural specification (MAS), protocol documentation, RTL files, existing SVAs, verification plans, timing diagrams, FSM state transition tables and project-specific macros and terminology. While generating outputs, the LLMs can refer to all these documents to enhance accuracy and decrease unnecessary speculative results. Here, we have used Claude Sonnet 4.6 AI Model integrated in Linux terminal as well as with Visual Studio Code. Figure 4 shows a snapshot of how initial conversation with Claude happened during one of the experiments. The Prompt given is below:

“Go through the RTL and the MAS in the doc directory to check that the proper reset happens in RTL for the config_rst condition. Also analyze the MAS to generate the proper \$stable assumptions on the intended config inputs of the <module_name> module.”

```

[cdhakraborty@ip-10 ~]$ claude
Claude Code v2.1.128
Sonnet 4.6 · API Usage Billing
/prj/sc

Opus 4.7 xhigh is now available! · /model to switch

Go through the RTL and the MAS in the doc directory to check that the proper reset happens in RTL
assumptions on the intended config inputs of the mc_ module.

• I'll explore the directory structure to find the relevant RTL files and MAS documentation before
Searched for 1 pattern, listed 1 directory (ctrl+o to expand)

• Good, I can see the MAS for _mc_ . Let me read it along with the RTL source files.
Searched for 1 pattern, read 1 file (ctrl+o to expand)

• Let me read the RTL source file and the generated RTL:
• Searching for 3 patterns, reading 1 file... (ctrl+o to expand)

• Reticulating... (18s · 1.6k tokens · still thinking)

> |
esc to interrupt

```

Figure 4: Claude prompt interaction in Linux exposed to MAS and RTL.

3.3 Agentic AI Architecture for Formal Verification

The third level extends the methodology from artifact generation to workflow orchestration (Figure 5). A skill is a structured prompt document that defines a repeatable AI-agent workflow. We can think of it as a reusable recipe that tells Claude:

what task to perform (e.g., generate microarchitecture docs from RTL), how to do it step-by-step, what inputs are needed and what outputs to produce, where to write files etc. When we invoke a skill, Claude reads the SKILL.md and executes the defined workflow using its own tools (file reads, bash commands, web fetches, etc.).

The proposed architecture can be elaborated in the following sub points:

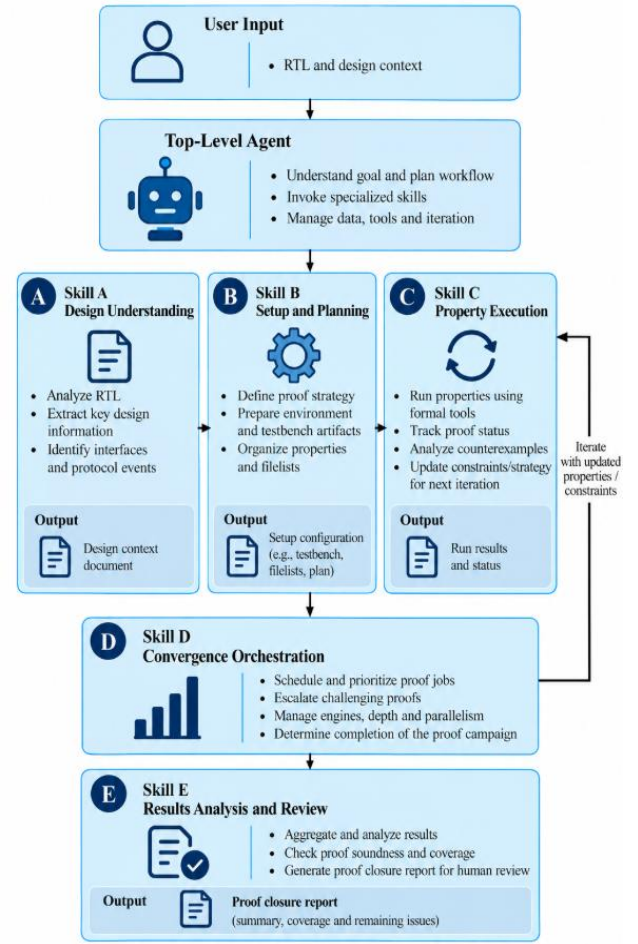


Figure 5: Agentic AI Architecture for Formal Verification

3.3.1 RTL Documentation: Skill A automatically generates uarch.md from RTL source files. Acts as the foundation for onboarding, builds the signal catalog, identifies protocol events, and formulates the environment model without re-reading raw RTL.

```

SAFETY PROPERTIES TO PROVE:
REQ-001 [safety]: mc_
REQ-002 [safety]: ear
REQ-003 [safety]: mc_
REQ-004 [safety]: mc_
REQ-005 [safety]: Pat
REQ-006 [safety]: Pat
REQ-007 [safety]: Pat
REQ-008 [safety]: Pat
REQ-009 [safety]: Pat

LIVENESS PROPERTIES TO PROVE:
REQ-007 [liveness, bound=unbounded/fair]: every
REQ-008 [liveness, bound=unbounded/fair]: every
REQ-009 [liveness, bound=unbounded/fair]: every

ENVIRONMENT CONSTRAINTS (assumes - primary inputs only, no RTL ou
mc_
mc_
mc_
mc_
cfg
mer
out

COVER GOALS:
COV-001: I
COV-002:
COV-003:
COV-004: mc_

Are these correct and complete? Any corrections before I proceed

```

Figure 6: Snapshot of Agentic AI generated Verification Goals

3.3.2 Setup prior to FV runs: Skill B is responsible for automatic creation of `design_model.yaml` that is the master handoff artifact from setup to the property proving loop phase. It tells Skill C what to prove, in what order, at what depth, and with which engine. Also consumed by Skill D for scheduling and Skill E for signoff classification. It also creates a human-readable version of the proof strategy, not consumed programmatically -purely for human reviewers. Alongside, it produces a `.fc` filelist to be consumed by `make runfv` to build the `.gf` compilation filelist. Finally, it creates the `fv_tb` which contains the primary SystemVerilog assertions, assumptions and covers and necessary helper code for reference model building.

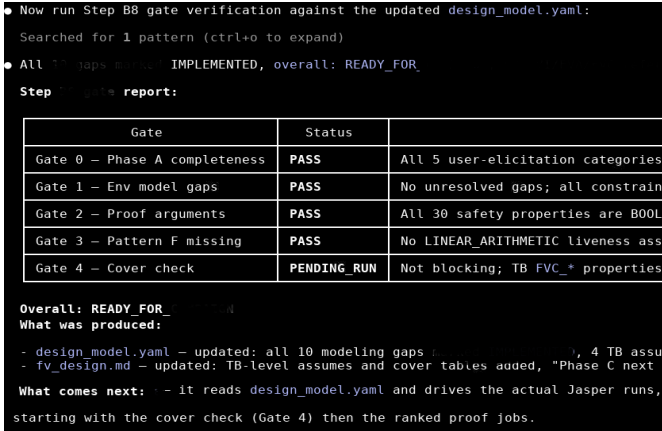


Figure 7: Snapshot of Agentic AI Flow showing automated verification and readiness to start proof jobs.

3.3.3 Property run loop: Skill C produces `state.yaml` which contains the live state machine of the verification tracking real-time proof status of every property. Skill D reads it to decide whether to escalate, advance, or complete. Skill E reads it for final signoff verification. Additionally it generates a logbook which provides full traceability -why each assume was added, what bug each counterexample (CEX) exposed, and whether fixes were justified. Referenced by Skill E during signoff. A `tel` file, acting as the sole interface between Skill C and FV tool is auto-generated as a scaffold on the first run; subsequently appended by Skill C. Per-property proof results get dumped into a `.rpt` file which gets parsed to classify each property result, extract CEX bounds, and identify which engine proved or failed each property -driving the next iteration of Skill C.

3.3.4 Convergence Orchestration: Skill D reads `proof_jobs[].rank` to determine parallel vs sequential job execution, escalates stuck jobs by increasing proof depth or switching engines, gates advancement to the next rank until all current rank jobs pass, terminates the loop when all ranks complete and triggers Skill E.

3.3.5 Soundness Review: Skill E generates proof closure report after reading `design_model.yaml`, `state.yaml`, and `logbook`. It is the final deliverable that exits the FV flow. It provides a rigorous and auditable basis for formal signoff.

4. Evaluation Methodology

Four verification modes are evaluated in this study, progressively shifting automation and context availability from fully manual to fully agentic. Manual FV is entirely engineer-driven, LLM-based generation automates artifact creation with prompt-provided context, and LLM with Retrieval further grounds generation in retrieved specification and RTL knowledge. Agentic AI represents the most advanced mode combining retrieved design context with reusable FV skills to autonomously handle generation, tool invocation, failure analysis, and iterative refinement, reducing the human role to intent, soundness, and final sign-off.

Mode	Context Available	Automation	Human Role
Manual FV	Specification + RTL + existing FV knowledge	Tool execution is engineer-driven	End-to-end
LLM	Prompt-provided context	Artifact generation	Generation review + execution + debug
LLM + Retrieval	Specification, RTL, protocol and FV knowledge retrieved at generation time	Context retrieval + artifact generation	Validation + execution + debug
Agentic AI	Retrieved design context + reusable FV skills + tool results	Generation + tool invocation + failure analysis + iterative refinement	Intent, soundness and final sign-off

TABLE I. Verification modes evaluated in the study

4.1 Evaluation Metrics

Seven metrics are used to evaluate the workflows. The first five characterize the quality and completeness of AI-generated verification artifacts, while the final two quantify engineering effort and schedule impact.

Metric	Definition	Measurement
Syntactic correctness rate	First-attempt compile success rate of generated formal artifacts.	First-attempt compilable artifacts / total generated artifacts × 100.
Requirement coverage rate	Fraction of applicable specification requirements captured by the generated verification artifacts.	Requirements correctly represented / applicable requirements × 100.
Semantic accuracy	Correctness of generated properties relative to a	Semantically correct artifacts / evaluated artifacts × 100.

	manually validated ground truth.	
Hallucination rate	Generated behavior for which no supporting basis can be identified in the specification, RTL, protocol context, or approved verification knowledge.	Unsupported generated behaviors / evaluated generated behaviors $\times 100$.
Corner-case coverage rate	Coverage of boundary and exceptional conditions identified from the specification.	Relevant corner cases captured / identified corner cases $\times 100$.
Human intervention count	Number of manual corrections or intervention events required before acceptable proof closure.	Count of documented interventions across the verification lifecycle.
Time to proof closure	Elapsed engineering time from specification analysis to closure of the defined proof objectives.	Measured elapsed time for each design/workflow combination.

TABLE II. Evaluation Metric Definitions

5. Observations

Case studies on multiple designs of variable size and complexity, 3 of which are included in this section.

5.1 Spec to Proof Closure Time Study

Case study 1: Connection Handshake Design (Low Complexity)

The Connection Handshake design implements a request-acknowledge protocol between two communicating design blocks, ensuring valid data transfer only when both sender and receiver are simultaneously ready, with key FV focus on handshake protocol correctness. This design served as the low-complexity baseline, with manual verification requiring approximately 24.3 hours across all phases. Standalone LLM reduced total effort by 87.3%, performing strongly on a structurally simple design. LLM+RAG further improved effort reduction to 94.1%, with retrieval context enabling near-instantaneous spec analysis and access to project-specific assertion templates. The Agentic AI achieved the highest reduction at 96.6% through autonomous compile-fix-recompile loops. Low-complexity designs therefore provide a practical starting point for Agentic AI adoption, where the verification workflow is sufficiently structured for autonomous orchestration while still providing measurable productivity gains.

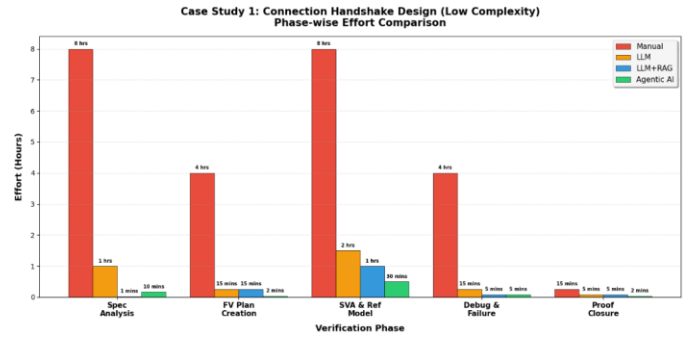


Figure 8: FV effort reduction across verification stages for the Connection Handshake (Simplest) design -Manual vs. AI-assisted approaches.

Case study 2: FSM + MUX Paired (Medium Complexity)

FSM + MUX Paired Design implements a 3-state Finite State Machine (FSM) governing state transitions driven by input conditions and control signals. A configurable Multiplexer (MUX) switches between multiple sources sending variable length packets and selects correct output based on FSM state encoding. Key FV focus: state reachability, illegal state transitions, output correctness per state, MUX selection consistency. Manual verification consumed approximately 60 hours. The standalone LLM achieved a 62.9% reduction but notably struggled with debug and proof closure phases, where effort remained unchanged from manual, indicating that LLM-generated assertions required extensive manual correction. LLM+RAG improved to 78.7%, with spec analysis collapsing from 12 hours to 2 minutes, though debug and proof closure phases showed no improvement over standalone LLM. The Agentic AI demonstrated the most significant leap at 94.1%, reducing debug time and hence proving to be advantageous when verification requires repeated interactions and code refinements.

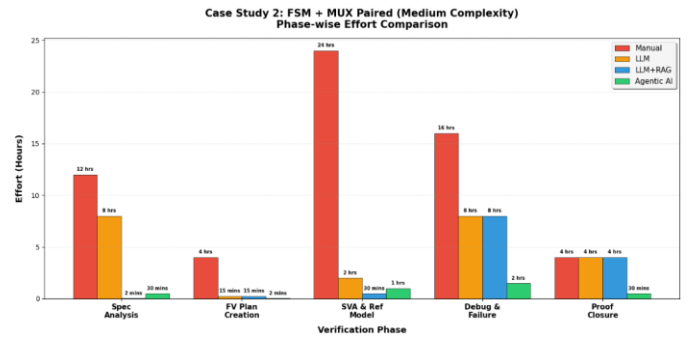


Figure 9: FV effort reduction across verification stages for the FSM+ MUX paired (Moderately Complex) design -Manual vs. AI-assisted approaches.

Case study 3: Multicast Response Collector (Medium to High Complexity)

The Multicast Response Collector is the most complex of the three designs, collecting and aggregating responses from multiple destination nodes for a single multicast transaction, tracking response completions across all targets before signaling done, and handling out-of-order and simultaneous responses, with key FV focus on no dropped or duplicate responses and correct aggregation logic. Manual verification required 376 hours. The standalone LLM achieved a 64.9% reduction, with SVA model generation still consuming a significant time. LLM+RAG improved to 76.6%, with spec analysis dropping dramatically through direct retrieval of protocol specification, though SVA development and debug phases remained substantial as testbench (TB) needed multiple prompts for updates of the fairly complex transaction tracker. Agentic AI achieved 86.8% reduction through iterative generation-proof-refinement cycles where the manual effort was just to review and give permissions to the suggested edits in the TB. The comparatively lower Agentic AI reduction at this complexity reflects the fact that authoring the exact model for such a design logic handling out of order multi cycle completions has a good scope of improvement over diminishing the number of iterative agentic refinement cycles.

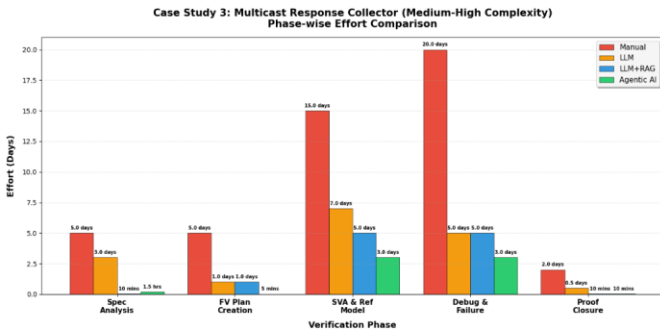


Figure 10: FV effort reduction across verification stages for the Multicast Response Collector design -Manual vs. AI-assisted approaches.

5.2 Observed Metric Values

Experimental results from the three case studies reveal that semantic accuracy, corner case coverage, and hallucination rate were most adversely impacted by raw LLM usage, as these metrics demand contextual understanding beyond pattern-based generation. Syntactic correctness showed the least degradation, with typo-free machine generation offsetting construct-level errors (eg. mixing blocking and non-blocking statements, procedural if-else inside property etc.). The Agentic AI configuration consistently matched manual quality across all metrics at a fraction of the time and intervention, confirming that retrieval-based grounding combined with iterative self-

correction through formal proof feedback represents the most effective paradigm for LLM-assisted formal verification.

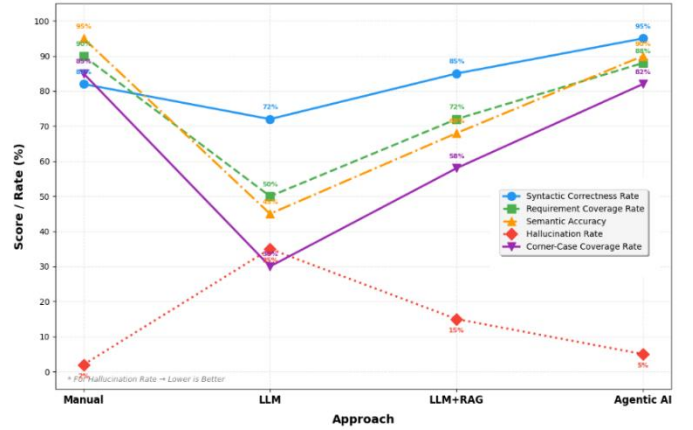


Figure 11: Formal Verification Quality Metrics Comparison Manual vs LLM vs LLM+Retrieval vs Agentic AI

6. AI Adoption Limitations

Although the evaluated AI-assisted workflows demonstrate significant productivity improvements, several limitations remain. Firstly, the experiments are conducted on a limited set of representative design blocks ranging from low to medium-high complexity. The observed productivity improvements may therefore not directly generalize to large production-scale SoCs with substantially larger state spaces, complex clock-domain interactions, or highly coupled protocol behavior.

Secondly, the quality of AI-generated verification artifacts remains dependent on the quality and completeness of the available design context. Ambiguous specifications, undocumented design behavior, incomplete RTL context, or inconsistent project documentation can lead to incorrect properties or inappropriate assumptions even when the generated artifacts are syntactically valid. In one of our initial observations, when mentioned about the input being ack and output being req for a particular block, Claude was asked to generate same cycle assumption that no spurious ack without a req. It gave assumption on output like `ack |-> req`. The FV engineer identified and corrected it as per the intent in the desired form like `!req |-> !ack`. Moreover, as observed across all three case studies, while AI accelerates initial generation, the SVA & Reference Model stage still demands considerable human effort to prune, review, and validate generated properties.

Thirdly, Agentic AI systems require tight integration with formal verification tools to programmatically interpret proof results, counterexamples, and vacuity witnesses.

7. Conclusion

This paper demonstrated that AI-assisted Formal Verification delivers consistent and measurable gains in productivity and verification quality across the complete FV lifecycle. Evaluated across three design case studies : Connection Handshake, FSM + MUX Paired, and Multicast Response Collector, results reveal a clear progression as AI assistance matures from standalone LLMs to fully Agentic orchestration. Experimental results across all seven metrics confirm that standalone LLMs offer speed but sacrifice quality in the absence of project context, with semantic accuracy, corner case coverage, and hallucination rate emerging as the most critically affected metrics. Retrieval-augmented context recovered quality by grounding generation in actual specification content, while Agentic AI approached near-manual verification quality - autonomously orchestrating property generation, failure analysis, and proof closure with minimal human intervention. Across all three designs, the methodology achieved at least 70% reduction in overall verification effort. Together, these results establish that the synergistic combination of Generative AI, retrieval-based context, and Agentic AI represents a transformative and scalable path toward democratizing Formal Verification in industrial SoC development.

References

- [1] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, Z. Xie, and H. Zhang, “AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs,” *arXiv preprint arXiv:2402.00386*, 2024. <https://arxiv.org/abs/2402.00386>
- [2] J. Blocklove, S. Garg, R. Karri, and H. Pearce, “Chip-Chat: Challenges and Opportunities in Conversational Hardware Design,” *arXiv preprint arXiv:2305.13243*, 2023. <https://arxiv.org/abs/2305.13243>
- [3] D. N. Gadde, A. Kumar, T. Nalapat, E. Rezunov, and F. Cappellini, “All Artificial, Less Intelligence: GenAI through the Lens of Formal Verification,” *arXiv preprint arXiv:2403.16750*, 2024. <https://arxiv.org/abs/2403.16750>
- [4] V. Pulavarthi, D. Nandal, S. Dan, and D. Pal, “Are LLMs Ready for Practical Adoption for Assertion Generation?” *arXiv preprint arXiv:2502.20633*, 2025. <https://arxiv.org/abs/2502.20633>
- [5] Z. He, Y. Pu, H. Wu, T. Qiu, and B. Yu, “Large Language Models for EDA: Future or Mirage?” *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, Art. no. 90, pp. 1–53, Oct. 2025, doi: 10.1145/3736167. <https://doi.org/10.1145/3736167>
- [6] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2nd ed. Boston, MA, USA: Springer, 2004.
- [7] B. Cohen, *SystemVerilog Assertions Handbook*, 4th ed. Santa Clara, CA, USA: VhdlCohen Publishing, 2016. <http://systemverilog.us/>
- [8] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Boston, MA, USA: Springer, 2006.
- [9] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2017, Feb. 2018.
- [10] Cadence Design Systems, “JasperGold Verification Platform.” https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html
- [11] Synopsys, “VC Formal Verification User Guide.” [Online]. Available: <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>
- [12] H. Ren *et al.*, “ChipNeMo: Domain-Adapted LLMs for Chip Design,” *arXiv preprint arXiv:2311.00176*, 2023. <https://arxiv.org/abs/2311.00176>
- [13] C. Weng, “LLM Powered Autonomous Agents,” Microsoft Technical Blog, Jun. 2023. <https://lilianweng.github.io/posts/2023-06-23-agent/>