FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures

Yunsheng Bai, Ghaith Bany Hamad, Chia-Tung Ho, Syed Suhaib, Haoxing Ren
NVIDIA
{yunshengb, gbanyhamad, chiatungh, ssuhaib, haoxingr}@nvidia.com

***Abstract*—Debugging formal verification (FV) failures represents one of the most time-consuming bottlenecks in modern hardware design workflows. When properties fail, engineers must manually trace through complex counter-examples spanning multiple cycles, analyze waveforms, and cross-reference design specifications to identify root causes—a process that can consume hours or days per bug. Existing solutions are largely limited to manual waveform viewers or simple automated tools that cannot reason about the complex interplay between design intent and implementation logic. We present FVDebug, an intelligent system that automates root-cause analysis by combining multiple data sources—waveforms, RTL code, design specifications—to transform failure traces into actionable insights. Our approach features a novel pipeline: (1) *Causal Graph Synthesis* that structures failure traces into directed acyclic graphs, (2) *Graph Scanner* using batched Large Language Model (LLM) analysis with for-and-against prompting to identify suspicious nodes, and (3) *Insight Rover* leveraging agentic narrative exploration to generate high-level causal explanations. FVDebug further provides concrete RTL fixes through its *Fix Generator*. Evaluated on open benchmarks, FVDebug attains high hypothesis quality and strong Pass@k fix rates. We further report results on two proprietary, production-scale FV counterexamples. These results demonstrate FVDebug's applicability from academic benchmarks to industrial designs.**

I. INTRODUCTION

Formal Verification (FV) is a cornerstone of modern VLSI design, using mathematical methods to prove that hardware designs adhere to their specifications [10, 11]. These specifications are captured through formal properties—often written as SystemVerilog Assertions (SVAs) [27]—that express expected design behaviors, such as "a request must be acknowledged within 3 cycles" or "the FIFO should never overflow." When a design violates such a property, industry-standard model checkers like Jasper [25] and VC Formal [23] generate a Counter-Example (CEX): a concrete execution trace showing cycle-by-cycle signal values that demonstrate the violation. This CEX, typically visualized as a waveform, provides an explicit failure scenario revealing where expected and actual behavior diverge.

However, deciphering a CEX is a notoriously manual and time-consuming task, with debugging consuming nearly 50% of verification engineers' time—their single largest activity [8, 16]. Engineers must trace signal dependencies backward through waveforms, cross-reference RTL logic, consult specifications, and synthesize this information to identify root causes—a process requiring deep understanding of design intent and implementation [15] that can stall development for hours or days. Commercial tools excel at waveform visualization [24, 25] but offer little automated reasoning; the burden of causal analysis remains manual.

Recent LLM-based approaches, while promising, have not yet captured the nuances of this workflow. Many are adapted from software debugging and fail to address hardware-specific concepts like cycle-accurate timing [5, 17, 22], while others rely on simplistic prompting strategies on raw trace files [13, 14]. In our empirical observations, these methods often produce false positives—flagging benign behaviors as suspicious—or miss true root causes by failing to trace multi-cycle causal chains.

We hypothesize that an effective automated debugger must emulate the structured, multi-source reasoning process of a human expert. Instead of treating the CEX as a flat sequence of events, it must first be structured into a representation that captures causality explicitly. We propose building a **Causal Graph** from the failure trace, where nodes represent signal events ("signal@cycle=value") and directed edges represent their immediate causal dependencies. This structured "mental model" of the failure enables systematic tracing of failure chains—mirroring how engineers mentally traverse waveforms backward to identify root causes.

We introduce **FVDebug**, the first *end-to-end* automated system that (i) builds such a causal mental model and (ii) exploits it through an LLM pipeline deliberately mirroring how verification engineers work:

1. **Graph Scanner** acts like an engineer's quick "sanity sweep," scanning every level of the causal graph. Through a context retriever that dynamically fetches relevant RTL code snippets and specification excerpts, the scanner evaluates each signal's behavior against both its implementation and intended functionality. A novel *for-and-against*

prompting scheme compels balanced evaluation—forcing the LLM to weigh evidence on both sides before flagging suspicious behavior.

2. **Insight Rover** plays the role of the engineer's deep dive: using the suspicious nodes from the scanner as initial seeds, it begins an agentic search of the causal graph. At each step, the LLM is presented with candidate neighboring nodes and autonomously selects which paths to pursue based on their relevance to forming coherent failure hypotheses. It generates and iteratively refines multiple competing hypotheses, using the context retriever to back them with cycle-accurate evidence and assigning confidence scores to converge on the most plausible root cause.

3. **Fix Generator & Report** synthesizes concrete RTL patches and produces comprehensive human-readable reports containing ranked hypotheses, causal timelines, and diff-ready code suggestions. This structured output report enables verification engineers and RTL designers to collaborate effectively with a shared, precise understanding of the failure mechanism, replacing the fragmented debugging discussions currently scattered across communication channels or email threads.

The main contributions of this paper are: (1) FVDebug is, to our knowledge, the **first realistic debugger** that automates the *entire* FV debug loop—from CEX to validated patch—while closely mimicking industrial engineering practice. (2) We introduce novel techniques for FV debugging including causal graph synthesis from counter-examples, for-and-against prompting for balanced signal analysis, and agentic narrative exploration that emulates how human engineers progressively refine hypotheses. (3) We develop a complete pipeline from failure trace to human-readable reports containing ranked root cause hypotheses, supporting evidence, causal chain timelines, and concrete RTL fixes—enabling effective collaboration between verification engineers and RTL designers. (4) On 38 real hardware failures, FVDebug achieves 95.6 % hypothesis quality for root cause identification, 71.1 % *Pass@1* and 86.8 % *Pass@5* fix rates. We also showcase FVDebug on failures found in real-world larger designs.

II. Related Work

A. *LLM/AI-Driven Debugging* without *Waveforms*

Static repair pipelines such as LLM-HDL leverage retrieval-augmented prompts to locate and patch functional RTL bugs [20]. VeriDebug couples contrastive embeddings with generative edits to unify localisation and fixing [29]. Beyond RTL, HLSDebugger adapts encoder–decoder models to high-level synthesis code, substantially improving logic-bug repair [28]. While effective on code-visible faults, these approaches [9, 12, 21, 31] lack temporal reasoning and cannot analyse failures that only manifest in execution traces.

B. *LLM/AI-Driven Debugging* with *Waveforms*

Trace-centric methods incorporate assertion failures or counter-examples directly into LLM prompts. AssertSolver learns from contrasting "right vs. wrong" traces to diagnose simulation-time assertion failures [33]. GenAI-Induction proposes helper invariants that unblock formal k-induction [13]. The multi-agent framework Saarthi iteratively proves properties and analyses failing CEXs in a closed loop [14]. These techniques typically still treat waveforms as flat text, limiting root-cause fidelity on multi-cycle failures; our work tackles this via explicit causal graphs.

C. *Commercial and Industrial Debug Solutions*

Mainstream EDA platforms—Cadence *Indago*, Synopsys *Verdi*, Cadence *JasperGold*—provide rich waveform viewers and coverage dashboards but leave causal reasoning to engineers [24–26]. Several start-ups now advertise AI-driven RTL debug, while general-purpose coding copilots showcase agentic software-debug workflows [1–3, 7]. Technical details remain scarce, and adapting these systems to waveform-centric hardware failures is non-trivial—highlighting the need for deeper, graph-structured research.

III. Methodology

A. *Problem Setup and System Overview*

When formal properties fail verification, model checkers produce counter-examples as flat signal sequences over time, obscuring causal relationships. FVDebug automates debugging by transforming these unstructured traces into structured causal graphs for systematic root cause analysis. Given a failing property, RTL design, and counter-example trace, our goal is to automatically identify root causes and generate fixes. Figure 1 illustrates our four-stage pipeline.

B. *Causal Graph Synthesis*

We transform counter-example traces into a causal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where nodes $\mathcal{V}$ represent signal events (signal, cycle, value) and edges $\mathcal{E}$ represent causal dependencies.

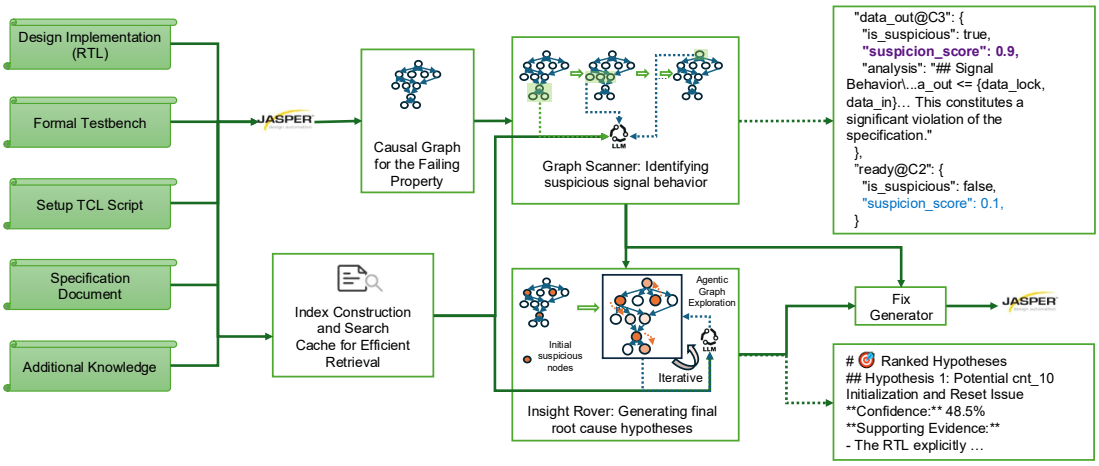


Figure 1: Overview of FVDebug. The system transforms formal verification counter-examples into actionable debugging insights through four stages: (1) Causal Graph Synthesis builds a directed acyclic graph capturing signal dependencies, (2) Graph Scanner performs efficient batched analysis to identify suspicious nodes, (3) Insight Rover explores competing hypotheses through intelligent graph navigation, and (4) Fix Generator produces concrete RTL patches and debugging reports. "C" denotes cycle in the example output of Graph Scanner.

Starting from the failing property at the violation cycle, we recursively query Jasper's `visualize -why` command to identify causal parents. For instance, querying `ready_add@5=0` reveals dependencies on `valid_out@5=0` and `valid_in@5=1`, establishing directed edges. We parameterize the construction with a trace depth (default: 20 cycles) that controls how far back we trace dependencies.

The recursive construction produces a tree with duplicate nodes from reconverging paths. We consolidate into a DAG by merging nodes with identical (signal, cycle, value) tuples, preserving all causal paths while preventing redundant analysis.

C. Graph Scanner

The Graph Scanner evaluates each node in $\mathcal{G}$ to identify suspicious behaviors. To optimize performance, it first pre-fetches RTL snippets, specifications, and documentation for all unique signals, maintaining a two-level cache (global design-wide and node-specific). The core of the scanner is our novel *for-and-against prompting* technique. For each node, the LLM must provide: (1) minimum two arguments FOR suspicion (timing issues, spec mismatches), (2) minimum two arguments AGAINST (valid patterns, expected behavior), (3) balanced conclusion with suspicion score (0.0-1.0), and (4) root cause vs symptom classification. This overcomes confirmation bias, where an LLM might simply verify that RTL logic matches its implementation without questioning the logic's correctness. For efficiency, nodes are processed in topological order using token-aware batching, with a binary search determining the maximum batch size that fits within a per-prompt token budget.

D. Insight Rover

The Insight Rover transforms individual suspicious nodes into coherent failure narratives using an agentic approach inspired by tree-of-thought reasoning [32]. For each suspicious node, an initial hypothesis is created. The system maintains multiple competing hypotheses (a minimum of 3, expanding as needed) and explores the causal graph to gather evidence for each. Instead of a brute-force search, the LLM agent intelligently selects which nodes on the exploration frontier to investigate next, drastically reducing the search space. After exploration converges or a pre-defined maximum number of iterations is reached, a final ranking of hypotheses is performed via the LLM based on sufficiency, evidence quality, mechanistic clarity, actionability, and narrative coherence.

E. Fix Generator with Ensemble Strategies

The Fix Generator performs ensemble diversity [6, 30] through five strategies varying context emphasis: *full context* (comprehensive but potentially noisy), *suspicious focus* (signals with score > 0.7), *narrative focus* (top hypothesis only), *minimal context* (surgical fixes), and *best-of meta-strategy* (reviews all outputs).

Each strategy generates fixes independently with retry mechanisms. Multi-level validation checks exact substring matching, handles whitespace normalization, and performs structural pattern alignment against RTL codebase $\mathcal{R}$. Fix

deduplication uses normalized hashing of (buggy_code, corrected_code) pairs, with consensus scoring: fixes from k of 5 strategies receive an increase in the confidence score by $\min(0.2k, 0.6)$.

Beyond patches, FVDebug produces structured reports containing: ranked hypotheses with evidence and confidence scores, cycle-by-cycle causal timeline, validated RTL fixes with diff-ready patches, and specification cross-references—enabling effective collaboration between verification and design teams.

IV. EXPERIMENTS

We evaluate FVDebug on the SVA-Eval-Human benchmark [33], a curated collection of 38 real hardware debugging challenges with human-verified ground truth fixes. Our implementation interfaces Python with Cadence Jasper 2023.12 through TCL commands. In addition, we evaluate FVDebug on two challenging CVA6 RISC-V processor failures originally discovered in AutoSVA [19], and we also report results on proprietary, production-scale FV counterexamples (design details masked).

A. Experimental Setup

The SVA-Eval-Human benchmark comprises diverse hardware designs with formal verification failures, ranging from simple arithmetic units to complex system-level modules. Each design includes a counter-example trace, the buggy RTL code, and the ground truth fix verified by human experts. We refer readers to [33] for detailed design descriptions and failure characteristics.

A.1 Baselines and Ablations

We evaluate the following methods, using o3-mini [18] as the underlying language model for all approaches:

Unstructured Baselines: (1) **Direct LLM**: Provides the counter-example trace, RTL code, and failing property directly to the LLM, asking it to identify the root cause and generate fixes. This represents the naive approach without any structural analysis or specialized prompting. (2) **Flat Trace Analysis**: Parses the counter-example into a structured chronological format, presenting signals and their values over time, but without constructing the causal graph. The LLM analyzes this temporal sequence to identify issues. This baseline isolates the value of causal structure versus mere temporal organization.

Component Ablations: (1) **FVDebug w/o For-Against**: Replaces our balanced evaluation prompting with standard bug-finding prompts that only ask for suspicious behaviors without requiring counterarguments. (2) **FVDebug w/o Rover**: Uses the Graph Scanner to identify suspicious nodes but skips the Insight Rover's narrative construction phase. Fixes are generated directly from the scanner's suspicious node analysis without exploring causal narratives. (3) **FVDebug w/o Ensemble**: Leverages only a single fix generation strategy (specifically, the full_context strategy) rather than our ensemble approach with multiple prompting perspectives.

A.2 Evaluation Metrics

First, we assess the quality of generated root cause hypotheses through LLM-based evaluation metrics that compare hypotheses against ground truth: (1) **Quality@Best**: Evaluates the absolute quality of the best hypothesis generated, regardless of ranking. (2) **NDCG@5**: Measures ranking quality by comparing hypothesis scores against ground truth relevance, with higher-ranked correct hypotheses contributing more. (3) **MRR**: Captures the average reciprocal rank at which the first relevant hypothesis appears. (4) **Kendall's τ**: Quantifies correlation between predicted rankings and ground truth quality.

Second, we measure functional correctness through Pass@k metrics, where proposed fixes are validated by applying them to the RTL and re-running formal verification in Jasper to check if the original assertion passes: (1) **Pass@1**: Measures first-attempt success rate for generated fixes. (2) **Pass@5**: Captures whether any of the top five fix suggestions resolves the failure.

B. Main Results

Table 1 presents comprehensive evaluation results across all methods. FVDebug achieves the highest Quality@Best score of 0.956, demonstrating superior root cause identification, while achieving the best fix generation performance with 71.1% Pass@1 and 86.8% Pass@5 rates.

C. Analysis of Results

The results reveal three critical insights. First, **causal structure is essential**: Flat Trace Analysis achieves only 0.474 Quality@Best despite temporal organization, while causal graph-based approaches achieve 0.795-0.956, demonstrating

Table 1: Comparison of FVDebug against baselines and ablations on the SVA-Eval-Human benchmark. Best results in **bold**.

	Root Cause Hypothesis Quality				Fix Generation	
Method	**Quality@Best**	**NDCG@5**	**MRR**	**Kendall's τ**	**Pass@1**	**Pass@5**
Unstructured Baselines						
Direct LLM	0.783	0.981	0.806	0.526	0.605	0.658
Flat Trace Analysis	0.474	0.948	0.804	0.511	0.632	0.816
Component Ablations						
FVDebug w/o For-Against	0.953	0.969	0.817	0.634	0.632	**0.868**
FVDebug w/o Rover	0.795	0.844	0.567	-0.176	0.643	0.821
FVDebug w/o Ensemble	**0.956**	**0.983**	**0.858**	**0.708**	0.684	0.763
FVDebug (Full)	**0.956**	**0.983**	**0.858**	**0.708**	**0.711**	**0.868**

that explicit causal relationships are fundamental to accurate root cause identification. Second, **narrative construction drives performance**: removing the Insight Rover causes dramatic degradation, confirming that connecting suspicious nodes into coherent causal chains is crucial. Third, **ensemble strategies improve robustness**: while single-strategy generation maintains hypothesis quality (0.956 Quality@Best), it achieves lower Pass@5 (0.763 vs 0.868).

D. Evaluation on Complex Processor Designs

To further assess FVDebug's capabilities on realistic hardware complexity, we evaluate on two challenging FV failures from the CVA6 RISC-V processor [19]. These failures, originally discovered in the AutoSVA project[1], represent real verification challenges encountered in production-grade processor designs: (1) **MMU Ghost Response**: A memory management unit (MMU) issue where misaligned requests trigger duplicate exception responses, violating transaction ID uniqueness properties. (2) **LSU Transaction ID Mismatch**: A load-store unit (LSU) failure where simultaneous exceptions and cache responses cause responses with untracked transaction IDs. Table 2 illustrates the substantial scale of these industrial designs.

Table 2: Scale and complexity metrics for CVA6 processor designs

Design	**RTL Files**	**Lines of Code**	**Graph Nodes**	**Graph Edges**	**Unique Signals**
CVA6 MMU	3	695	172	235	76
CVA6 LSU	7	1,918	170	239	122

Figure 2a shows a representative subgraph from the LSU failure's causal analysis, illustrating the complex signal dependencies FVDebug must navigate. This snippet traces backwards from the failing assertion through 5 levels of causality—a fraction of the full 20-level, 170-node graph.

Table 3 presents evaluation results. FVDebug achieves highest Quality@Best (0.713) despite the complexity, though absolute scores are lower than simpler benchmarks, reflecting the inherent difficulty of multi-module processor debugging. While automated fix generation for such multi-line issues remains future work, FVDebug's ability to navigate these massive causal graphs and identify root causes with 0.713 quality represents an advance for industrial verification workflows.

E. Industrial FV Case Studies (Proprietary Designs)

We evaluate FVDebug on two proprietary, production-scale FV counterexamples (CEX), with design details masked per policy. Signal names have been replaced with descriptive placeholders in angle brackets (e.g., `<signal_name>`) to protect intellectual property. For each case, an expert provides the ground truth root cause, which we use to verify the FVDebug's outputs.

CEX-1 (FV Testbench Error)

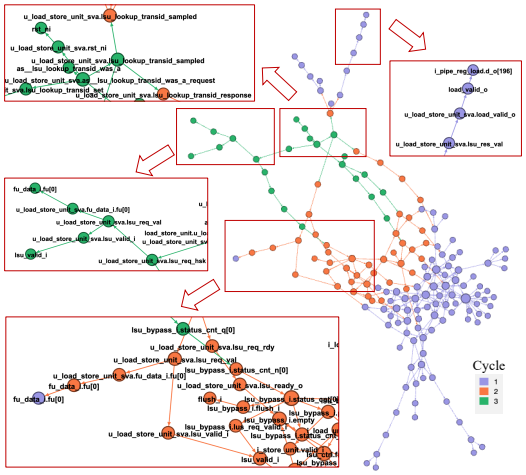
This high-difficulty case involves an incorrect internal token counter. FVDebug correctly identifies the root cause as a functional bug in the testbench logic.

FVDebug Root Cause: The `<internal_token_count>` calculation is incomplete—it only sums 4

[1] https://github.com/PrincetonUniversity/AutoSVA

```
as__lsu_lookup_transid_was_a_request (C:3, V:FAIL)
|-- lsu_lookup_transid_response (C:3, V:1'b1)
|   |-- lsu_res_hsk (C:3, V:1'b1)
|   |   `-- load_valid_o (C:3, V:1'b1)
|   |       `-- i_pipe_reg_load.d_i[196] (C:2, V:1'b1)
|   |           |-- i_load_unit.ex_i.valid (C:2, V:1'b1)
|   |           |   |-- i_mmu.misaligned_ex_q.valid (C:2, V:1'b1)
|   |           |   |   `-- i_mmu.misaligned_ex_n.valid (C:1, V:1'b1)
|   |           |   |       |-- i_mmu.lsu_req_i (C:1, V:1'b1)
|   |           |   |       |   |-- ld_translation_req (C:1, V:1'b1)
|   |           |   |       |   `-- lsu_ctrl.fu (C:1, V:LOAD)
|   |           |   |       `-- i_mmu.misaligned_ex_i.valid (C:1, V:1'b1)
|   |           |   |           |-- data_misaligned (C:1, V:1'b1)
|   |           |   |           |   |-- lsu_ctrl.operator (C:1, V:SD)
|   |           |   |           |   `-- lsu_ctrl.vaddr[2] (C:1, V:1'b1)
|   |           |   |           `-- lsu_ctrl.overflow (C:1, V:1'b0)
|   |           |   `-- i_mmu.pmp_data_allow (C:2, V:1'b1)
|   |           `-- i_load_unit.state_q[1:0] (C:2, V:2'b00)
|   `-- lsu_res_transid (C:3, V:3'b000)
|       `-- load_trans_id_o (C:3, V:3'b000)
|           `-- i_load_unit.load_data_q.trans_id (C:2, V:3'b000)
|-- lsu_lookup_transid_sampled (C:3, V:4'b0000)
|   |-- lsu_lookup_transid_response (C:2, V:1'b0)
|   |-- lsu_lookup_transid_sampled (C:2, V:4'b0000)
|   `-- lsu_lookup_transid_set (C:2, V:1'b0)
|-- lsu_lookup_transid_set (C:3, V:1'b0)
|   `-- lsu_req_hsk (C:3, V:1'b0)
|       |-- lsu_req_rdy (C:3, V:1'b0)
|       `-- lsu_req_val (C:3, V:1'b0)
|       |-- u_load_store_unit_sva.fu_data_i.fu[0] (C:3, V:1'b0)
|       |   `-- fu_data_i.fu[0] (C:3, V:1'b0)
```

(a) Partial causal graph from the CVA6 LSU failure. "C" and "V" refer to cycle and value, respectively.



(b) Visualization of the same failure's causal graph. The image is generated via Gephi [4].

Figure 2: CVA6 LSU failure: textual subgraph (left) and full-graph visualization (right).

Table 3: Comparison of FVDebug against baselines and ablations on CVA6 processor failures. Best results in **bold**.

	Hypothesis Quality Metrics			
Method	**Quality@Best**	**NDCG@5**	**MRR**	**Kendall's τ**
Unstructured Baselines				
Direct LLM	0.575	0.801	0.500	0.333
Flat Trace Analysis	0.475	0.800	0.531	0.100
Component Ablations				
FVDebug w/o For-Against	0.644	0.791	0.531	0.306
FVDebug w/o Rover	0.300	0.742	0.276	0.294
FVDebug (Full)	**0.713**	**0.875**	**0.667**	**0.371**

specific FIFO token counts but does NOT include the mesh input path (<mesh_in_signal>) where tokens are actually entering the design.

Expert evaluation confirms an "Excellent Match," noting that FVDebug correctly identifies the missing signal in the counter logic and understands the architectural flaw. The system pinpoints the specific functional error and recommends the correct fix, aligning perfectly with the expert's independent analysis.

CEX-2 (Missing Constraint)

This case involves a FIFO overflow caused by a missing input constraint. FVDebug diagnoses the issue as a failure in the backpressure mechanism.

> *FVDebug Root Cause:* Backpressure Mechanism Failure: The FIFO continues accepting write requests (<fifo_write_enable>=1) even when the FIFO is full (<fifo_full_status>=1). This is caused by an unconstrained input signal (<req_valid_signal>) driving write logic without proper backpressure handling.

This analysis is rated as a "Strong Match" by the expert. Although the terminology differs slightly (FVDebug's "backpressure failure" vs. the expert's "credit protocol violation"), the core diagnosis is identical. FVDebug successfully identifies the problematic input signal and recommends the correct solution: adding a constraint to the formal testbench. Table 4 summarizes the complexity and performance metrics for these case studies.

V. CONCLUSION AND FUTURE WORK

We presented FVDebug, an automated FV debugging system that transforms counter-examples into *causal graphs* and applies a multi-stage LLM pipeline—balanced scanning and agentic narrative exploration—to generate high-quality

Table 4: Industrial CEX complexity and FVDebug performance metrics (masked design details).

Case	RTL Files	Lines of Code	Unique Signals	Graph Nodes	Graph Edges	Graph Depth	Jasper Calls	LLM Calls	Total Runtime
CEX-1 (Testbench Error)	265	560,202	123	189	227	15	163	25	4m 12s
CEX-2 (Missing Constraint)	257	554,983	41	80	86	11	67	21	6m 07s

root-cause explanations and practical fixes. On open benchmarks, FVDebug achieves strong hypothesis quality and Pass@k rates. On proprietary, production-scale counterexamples, we demonstrate applicability by reporting graph- and code-scale metrics with expert-verified root causes.

Looking forward, the causal graph approach naturally extends to simulation-based verification, where graphs could be constructed from signals whose values mismatch golden references rather than from failed properties. This would unify debugging workflows across verification methodologies, providing consistent root cause analysis regardless of failure detection method.

ACKNOWLEDGEMENTS

The authors deeply thank **Prosenjit Chatterjee** and **Brucek Khailany** for their technical guidance and support. We are grateful to our collaborators at **Cadence Design Systems**, including **Paula Mathias**, **Tulio Leao**, **Craig Deaton**, **Matheus Fonseca**, **Dan Benua**, **Chris Komar**, **Gargi Sharma**, **Guilherme Fernandes Silvano**, **Luis Humberto Rezende Barbosa**, **Fabiano Peixoto**, **Kiran Joseph**, **Christopher Cronk**, **Shiva Borzin**, **Karam Abdelkader**, and **Andrew Penrose**, for extensive discussions on Jasper. We also extend our appreciation to NVIDIA colleagues **Sid Dhodhi**, **Vigyan Singhal**, **Sergei Posnov**, **Kalyan Krishnamani**, **Mark Bezdany**, **Scott Fields**, **Stuart Oberman**, **Jiaxin Wan**, **Teo Ene**, **Marcelo Orenes-Vera**, **Mukhdeep Benipal**, **David Nolte**, **Leo Qiu**, **Michael Miao**, **Cedar Huang**, **Chi Gu**, and many colleagues for their valuable contributions.

REFERENCES

[1] Alpha Design AI. Chipagents: Agentic AI for RTL design and debug. `https://chipagents.ai`, 2025. Accessed Jul. 2025.

[2] ChipStack AI. Chipstack – chip design reimagined. `https://www.chipstack.ai`, 2025. Accessed Jul. 2025.

[3] Anthropic. Claude code. `https://www.anthropic.com/claude-code`, 2025. Accessed Aug. 2025.

[4] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[7] Cursor. Cursor – the AI code editor. `https://cursor.com`, 2025. Accessed Aug. 2025.

[8] Harry D Foster. 2020 wilson research group functional verification study. Technical report, Siemens EDA, 2020.

[9] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. Llm4sechw: Leveraging domain-specific large language model for hardware debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2023.

[10] Orna Grumberg, EM Clarke, and D Peled. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany*, 1999.

[11] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2):151–238, 1992.

[12] Muhammad Hassan, Mohamed Nadeem, Khushboo Qayyum, Chandan Kumar Jha, and Rolf Drechsler. Prompt. verify. repeat. llms in the hardware verification cycle.

[13] Aman Kumar and Deepak Narayan Gadde. Generative ai augmented induction-based formal verification. In *2024 IEEE 37th International System-on-Chip Conference (SOCC)*, pages 1–2. IEEE, 2024.

[14] Aman Kumar, Deepak Narayan Gadde, Keerthan Kopparam Radhakrishna, and Djones Lettnin. Saarthi: The first ai formal verification engineer. *arXiv preprint arXiv:2502.16662*, 2025.

[15] Minh Luu, Surya Jasper, Khoi Le, Evan Pan, Michael Quinn, Aakash Tyagi, and Jiang Hu. Vcdiag: Classifying erroneous waveforms for failure triage acceleration. *arXiv preprint arXiv:2506.03590*, 2025.

[16] Ann Mutschler. Debug tops verification tasks, Dec 2018. Semiconductor Engineering.

[17] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *ICML*, pages 26106–26128. PMLR, 2023.

[18] OpenAI. OpenAI o3-mini. `https://openai.com/index/openai-o3-mini/`, January 2025. Accessed: 2025-08-26.

[19] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. Autosva: Democratizing formal verification of rtl module interactions. In *2021 58th ACM/IEEE DAC*, pages 535–540. IEEE, 2021.

[20] Khushboo Qayyum, Chandan Kumar Jha, Sallar Ahmadi-Pour, Muhammad Hassan, and Rolf Drechsler. Llm-assisted bug identification and correction for verilog hdl. *ACM Transactions on Design Automation of Electronic Systems*, 2025.

[21] Dipayan Saha, Shams Tarek, et al. Sv-llm: An agentic approach for soc security verification using large language models. *arXiv preprint arXiv:2506.20415*, 2025.

[22] Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro. Nemotron-cortexa: Enhancing llm agents for software engineering tasks via improved localization and solution diversity. In *Forty-second International Conference on Machine Learning*.

[23] Synopsys Inc. Vc formal. `https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html`, 2023.

[24] Synopsys, Inc. *Verdi Automated Debug System*, 2024. Online product page, accessed Jun 2025.

[25] Cadence Design Systems. *Cadence JasperGold Formal Verification Platform*, 2023. Version 2023.12.

[26] Cadence Design Systems. Indago debug platform – product brief. `https://dvcon-proceedings.org/wp-content/uploads/Indago%E2%84%A2-Debug-Platform-Overview.pdf`, 2025. Accessed Jul. 2025.

[27] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for SystemVerilog assertions*. Springer, 2005.

[28] Jing Wang, Shang Liu, Yao Lu, and Zhiyao Xie. Hlsdebugger: Identification and correction of logic bugs in HLS code with LLM solutions. In *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2025.

[29] Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. Veridebug: A unified llm for verilog debugging via contrastive embedding and guided correction. *ICLAD*, 2025.

[30] Xuezhi Wang, Jason Wei, et al. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[31] Ke Xu, Jialin Sun, et al. Meic: Re-thinking rtl debug automation using llms. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2024.

[32] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 36:11809–11822, 2023.

[33] Jie Zhou, Youshu Ji, Ning Wang, Yuchen Hu, Xinyao Jiao, Bingkun Yao, Xinwei Fang, Shuai Zhao, Nan Guan, and Zhe Jiang. Insights from rights and wrongs: A large language model for solving assertion failures in rtl design. *DAC*, 2025.