LLM-Assisted Analysis of On-Chip Protocol Implementations

Melisande Zonta-Roudes *ETH Zurich*
Nora Hinderling *ETH Zurich*
Supraja Sridhara *ETH Zurich*
Srinidhi Nagendra *MPI-SWS*
Shweta Shinde *ETH Zurich*

I. Introduction

Hardware designs comprise many IP blocks that must communicate over rigorously specified on-chip protocols. These protocols regulate not only functionality but also cycle-accurate timing and ordering. Even a single cycle deviation can become a permanent silicon fault with functional consequences (e.g., execution integrity, denial of service). To verify and validate these critical protocol implementations, existing tools use assertion-based verification (ABV) and formal methods. As input, these approaches require properties that need to be extracted from long, natural-language specifications and adapted to a given implementation-under-test. Currently, engineers perform this property generation manually, making it slow and error-prone, and hindering the ability to determine whether implementations adhere to specifications or exhibit violations.

II. Method

In this paper, we introduce a method to automate the manual extraction of properties from specifications in a manner that generalizes across protocols but can still be used to test specific protocol implementations. Recent advances in large language models (LLMs) enable understanding natural language and generating code, making them promising for generating properties from long specifications for specific implementations. However, using LLMs to generate such properties effectively is challenging: our experiments show that directly providing both specification documents and implementations leads to poor performance, including missed properties and hallucinations. This is not surprising as LLMs cannot handle and reason about large contexts, documents and implementations in our case.

This approach uses several insights based on iterative refinement of LLM usage to ensure that we capture specification rules while maintaining accuracy. First, instead of providing the entire specification at once, we observe that structuring the input and identifying key elements such as definitions and requirements improves the model's ability to capture relationships between signals and their temporal behavior. Second, introducing an intermediate representation that captures temporal aspects (e.g., timing, ordering, stability) helps guide the model and reduces hallucinations, while making the intent of generated properties more explicit. Lastly, incorporating implementation context when generating executable properties ensures that the model focuses on the relevant signals for each specification rule. More importantly, this enables explicit linkage between generated properties and their originating specification statements, which is essential for analyzing and resolving detected violations and reduces the manual effort required to assess assertion relevance.

TABLE I: Per-implementation results with property counts and violation breakdowns. ✓: proven, ✗: violated, ∅: unsupported; #violations vs. prior work.

Protocol	Impl.	#properties			#violations	
		✓	✗	∅	common	SIFT only
AXI	AX_{XFS}	133	90	20	37	53
	AX_{XFM}	140	77	26	33	44
	AX_{XLS}	39	36	168	25	11
	AX_{XLM}	45	30	168	26	4
	AX_{PFS}	34	39	170	28	11
AHB	AH_A	30	41	5	14	25
	AH_S	31	39	6	18	23
APB	AP	29	33	0	22	11
TileLink	TL	15	37	6	12	25
OBI	OB	96	14	33	–	–
Total		**531**	**436**	**582**	**215**	**207**

III. Evaluation

To demonstrate the effectiveness of this approach, we evaluate it across 5 different hardware protocols (AXI, AHB, APB, TileLink, OBI) with specification documents ranging from 28 to 132 pages and apply the generated properties to two assertion languages (SVA/PSL). Overall, the approach synthesizes 1,651 properties (including 501 unique, 273 absent from prior verification IPs), enabling broad coverage of protocol requirements.

We then apply these properties across ten real-world implementations (AMD Xilinx, OpenHW, lowRISC, PULP), where they expose 436 protocol violations with 215 that prior work couldn't uncover. Table I summarizes the split between proven, violated, and unsupported properties for the different implementations, highlighting differences in feature support and correctness. In particular, some implementations exhibit a high number of unsupported properties due to omitted features, while others show a larger fraction of violated properties, indicating deviations from specification intent.

From the identified violations, we further construct FPGA-based exploits that demonstrate their practical impact, showcasing vulnerabilities such as data leakage, execution integrity violations, and denial-of-service conditions.

IV. Conclusion

We introduce a specification-driven approach for generating implementation-specific verification properties across diverse protocols. Our evaluation reveals hundreds of protocol violations, including exploitable vulnerabilities.