Late Breaking Results: LLM-assisted Automated Incremental Proof Generation for Hardware Verification

Khushboo Qayyum[2], Muhammad Hassan[1,2], Sallar Ahmadi-Pour[1], Chandan Kumar Jha[1], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hassan, sallar, khushboo, chajha, drechsler}@uni-bremen.de

***Abstract*—In this paper, we propose a methodology for hardware verification assisted by *Large Language Models* (LLMs) in the incremental proof generation process. First, an LLM identifies the basic module of the *Design Under Verification* (DUV), followed by expanding the proof scope as more modules are added. LLMs assist in defining and verifying invariants for each module using the Z3 solver, and in formulating integration properties at module interfaces. Our case studies on a *Ripple Carry Adder* (RCA) and a *Dadda Tree Multiplier* (DTM) demonstrate that LLMs enhance the efficiency and accuracy of hardware verification.**

I. Introduction

In the ever-evolving landscape of hardware design, ensuring the reliability and correctness of hardware systems is a paramount concern. In this context, formal verification has been shown as a vital approach to mathematically guarantee the absence of bugs in hardware systems [1], [2]. Central to formal verification is the generation of functional properties, or invariants, for the *Design Under Verification* (DUV). Traditionally, this is a manual and time-intensive process, involving careful analysis and translation of complex natural language specifications into verifiable properties, a task that becomes more challenging as systems grow in scale and complexity.

This is precisely where the advent of *Large Language Models* (LLMs) like OpenAI's GPT-4 and Google's Gemini mark a transformative shift. LLMs, with their advanced natural language processing capabilities, can automate and streamline the property generation process. They excel in interpreting complex hardware specifications and converting them into precise formal representations, such as SystemVerilog Assertions (SVA) [3]–[5], *System-on-Chip* (SOC) properties [6]–[9], and stimuli generation [10]. This automation not only accelerates the verification process but also reduces the potential for human error in the initial stages of property creation. However, a significant challenge arises when the formal model of the DUV, combined with numerous properties, becomes overwhelmingly large. In such scenarios, the verification process, particularly the generation of proofs, becomes excessively slow, diminishing the efficiency gains made by LLMs in property creation.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project ECXL under contract no. 01IW22002, the project PaSVer under contract no. 16ME0855, the project SASPIT under contract no. 16KIS1852K and the project Scale4Edge under contract no. 16ME0127.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06...$15.00
http://dx.doi.org/10.1145/3649329.3663498

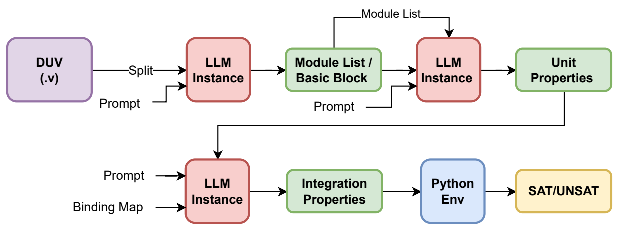

Fig. 1. Overview - LLM-assisted automated incremental proof generation

In this regard, the potential of LLMs in enhancing incremental proof generation presents an intriguing solution. Incremental proof generation simplifies the verification process by dividing it into smaller, manageable segments. This approach involves verifying each sub-module of the DUV individually, along with its interconnections, before methodically integrating these verified components to encompass the entire system.

In this paper, we present a methodology for formal hardware verification assisted by LLMs into the incremental proof generation process. It starts with an LLM identifying the basic module of the DUV, setting the stage for proof generation. The process then incrementally adds modules, expanding the proof scope progressively. Initially, the LLM defines the invariants, termed as *Unit Properties* (UPROP) for the basic module, and these are verified using the Z3 solver. As more modules are integrated, the LLM helps in formulating invariants at the interfaces, termed *Integration Properties* (IPROP). This step-by-step approach leads to a comprehensive proof for the entire DUV. Our two case-studies; a *Ripple Carry Adder* (RCA) and a *Dadda Tree Multiplier* (DTM), indicate that LLMs can significantly aid in proof generation and hardware verification.

II. LLM-assisted Methodology

In this section, we provide an overview of the proposed LLM-assisted verification methodology for automated and systematic incremental proof generation. The methodology comprises five stages as shown in Fig. 1, 1) code splitting, 2)

TABLE I
A SELECTION OF UPROP FOR 1-BIT FULL-ADDER IN NATURAL LANGUAGE AND Z3PY

Natural Language Description	Z3Py Representation
Zero Input Case: If all inputs (A, B, and Cin) are zero, then both the Sum and Cout outputs should be 0	`zero_input_case = And(Not(A), Not(B), Not(Cin), Not(Sum), Not(Cout))`
Two Inputs High (A and B are 1): If A and B are both 1, with Cin being 0, the Sum should be 0 and Cout should be 1.	`two_inputs_AB = And(A, B, Not(Cin), Not(Sum), Cout)`
All Inputs High: A, B, and Cin are all 1, Sum and Cout should both be 1.	`all_inputs_high = And(A, B, Cin, Sum, Cout)`

basic module search, 3) unit property generation, 4) encapsulating block selection, and 5) integration property generation.

In the first stage, an LLM instance processes Verilog files of the DUV, dividing them into smaller modules. This division helps manage the code more effectively and circumvents the LLM's token limit. In the second stage, the LLM identifies the simplest functional module (basic module) within the DUV, such as a half-adder. This module sets the stage for our incremental proof. The LLM also compiles a list of the module's functionalities and constructs a corresponding Z3Py model. In the third stage, the LLM generates a list of invariants using the identified basic module, termed UPROP, in natural language. These properties are then converted into Z3Py format by the LLM. The Z3 solver employs these properties and the module's Z3Py model to determine satisfiability - *SAT/UNSAT*. In stage four, a Verilog parser extracts *binding* information of modules, facilitating systematic proof extension to connected modules. The LLM formulates invariants for the interfaces between connected modules, termed IPROP. These properties are translated into Z3Py format and analyzed with the Z3 solver for satisfiability. In stage five, the process of stages 2, 3, and 4 repeats iteratively, expanding from the basic module to the top-level module of the DUV. If the Z3 solver finds a counter-example at any stage, the LLM is prompted to generate a corresponding Verilog test case for the testbench. The proposed methodology enables systematic and efficient verification of hardware designs, with the assistance of LLMs to streamline the incremental proof generation process.

III. EXPERIMENTAL EVALUATION

In this section, we discuss two case-studies using Z3 prover to generate proofs and OpenAI's GPT-4 as the LLM.

A. Case-study : Ripple Carry Adder

For this case study, a 4-bit RCA was used, which can extended to n-bit adder. For RCA, a full-adder is identified as a basic module and accordingly UPROP are generated. A selection of UPROP in natural language and Z3Py representation are shown in Table I. Consider the case when all inputs are high, i.e., if A, B, and C_{in} are all 1, then the *Sum* should be 1 and C_{out} should be 1 as well. This case is the full utilization of the adder where three $1s$ result in the binary value 11. The corresponding UPROP in Z3Py representation is: $all\_inputs\_high = And(A, B, C_{in}, Sum, C_{out})$. Similarly, the IPROP generated by stage four are shown in Table II.

TABLE II
IPROP OF TWO CONNECTED FULL ADDERS

Natural Language Description	Z3Py Representation
Sum Independence: The sum output of the first full adder (Sum1) is independent of the second full adder's inputs and operation	`sum_independence = Not(Exists([A2, B2, Cin2], Sum1 == Xor(A2, B2, Cin2)))`
No False Carry: If the first full adder does not generate a carry, the second full adder's carry-in should be zero	`no_false_carry = Implies(Not(Cout1), Not(Cin2))`
Carry Generation and Handling: If the first full adder generates a carry, the second full adder should process this carry along with its own inputs	`carry_generation_handling = Implies(Cout1, And(Cin2, full_adder_2_logic))`

B. Case-study : Dadda Tree Multiplier

For the second case-study, we applied our methodology to a 4-bit DTM that is extendable to n-bit multiplier. In this case, the LLM identified the single-bit multiplication units that form the foundation of the partial product matrix as a basic module. Afterwards, the proof was extended to include half adders and *Carry-select Adder* (CSA). Table III shows two UPROP for basic module of DTM and half adder. Finally, the proof connects all the modules.

TABLE III
CASE-STUDY: DADDA MULTIPLIER'S UNIT PROPERTIES

Natural Language Description	Z3Py Representation
AND Gate Functionality: When both inputs are 1, the output is 1; otherwise, the output is 0.	`def and_gate_functionality(a, b): return And(a == 1, b == 1)`
Half Adder - Summing without Carry: If only one input is 1 and the other is 0, the sum output is 1, and the carry output is 0.	`def half_adder_sum_without_carry(a, b): sum = Xor(a, b); carry = And(a, b); return And(sum == 1, carry == 0)`

IV. CONCLUSION

In this paper, we presented a methodology for hardware verification assisted by LLMs, which involves an incremental proof generation process. An LLM first identifies the basic module of the DUV, then expands the proof scope with additional modules. LLMs are crucial for defining and verifying module-specific invariants with the Z3 solver and developing integration properties at module interfaces. Case studies on a *Ripple Carry Adder* and a *Dadda Tree Multiplier* are demonstrated.

REFERENCES

[1] R. Drechsler, *Advanced formal verification*. Springer, 2004.
[2] ——, *Formal verification of circuits*. Springer Science & Business Media, 2013.
[3] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.
[4] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "From rtl to sva: Llm-assisted generation of formal verification testbenches," *arXiv preprint arXiv:2309.09437*, 2023.
[5] C. Sun, C. Hahn, and C. Trippel, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *First International Workshop on Deep Learning-aided Verification*, 2023.
[6] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "Llm for soc security: A paradigm shift," *arXiv preprint arXiv:2310.06046*, 2023.
[7] X. Meng, A. Srivastava, A. Arunachalam, A. Ray, P. H. Silva, R. Psiakis, Y. Makris, and K. Basu, "Unlocking hardware security assurance: The potential of llms," *arXiv preprint arXiv:2308.11042*, 2023.
[8] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," *arXiv preprint arXiv:2302.01215*, 2023.
[9] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "Llm-guided formal verification coupled with mutation testing."
[10] Z. Zhang, G. Chadwick, H. McNally, Y. Zhao, and R. Mullins, "Llm4dv: Using large language models for hardware test stimuli generation," *arXiv preprint arXiv:2310.04535*, 2023.