AssertionBench: A Benchmark to Evaluate Large-Language Models for Assertion Generation for Hardware Design

Vaishnavi Pulavarthi[1], Deeksha Nandal[1], Soham Dan[2*], Debjit Pal[1*]

[1]Dept. of Electrical and Computer Engineering,
University of Illinois Chicago, Chicago IL 60607,
[2]Microsoft,

Correspondence: dpal2@uic.edu

Abstract

Assertions have been the de facto collateral for hardware verification for over a decade. The verification quality, *i.e.*, detection and diagnosis of corner-case design bugs, is critically dependent on the assertion quality. There has been a considerable amount of research to generate high-quality assertions from hardware design source code and design execution trace data. With the recent advent of generative AI techniques such as Large Language Models (LLMs), there has been a renewed interest in deploying LLMs for assertion generation. However, there is little effort to quantitatively establish the effectiveness and suitability of various LLMs for assertion generation. In this paper, we present AssertionBench, a novel benchmark to evaluate LLMs' effectiveness for assertion generation quantitatively. AssertionBench contains 100 curated Verilog hardware designs from OpenCores and formally verified assertions for each design, generated from GoldMine and HARM. We use AssertionBench to compare state-of-the-art LLMs to assess their effectiveness in inferring functionally correct assertions for hardware designs. Our experiments comprehensively demonstrate how LLMs perform relative to each other, the benefits of using more in-context examples in generating a higher fraction of functionally correct assertions, and the significant room for improvement for LLM-based assertion generators.

1 Introduction:

System-on-Chip (SoC) designs are the building blocks for many safety-critical computing systems, including vehicular systems, infrastructure, military, and industrial automation. It is crucial to establish that the SoCs are functionally correct, safe, and secure to ensure that the SoC designs function as intended and are free from errors and vulnerabilities.

*SD and DP jointly supervised this work.

Assertions or *design invariants* are mathematical encoding (in Boolean logic) of desired design properties that should hold True for the hardware design. Assertion-based Verification (ABV) (Witharana et al., 2022) has long emerged as the de facto standard to verify the security and functional correctness of SoCs. However, crafting a succinct set of assertions for hardware designs is a tedious and time-consuming task, often requiring considerable human ingenuity. Too many assertions can negatively affect verification performance with a prolonged verification closure, whereas too few assertions may result in insufficient design coverage causing corner case design bugs to escape to production and mass manufacturing. ***Consequently, it is crucial to develop automated and scalable techniques to rapidly generate a succinct set of hardware design properties targeting design functionality, safety, and security***.

There has been a considerable amount of research work for assertion generation using two different paradigms – lightweight static analysis of design source code and formal verification (Tiwari et al., 2001; Păsăreanu and Visser, 2004), and data-driven statistical analysis, *e.g.*, data mining (Ernst et al., 2000; Hangal et al., 2005; Pinter and Majzik, 2005; DeOrio et al., 2009; Chang and Wang, 2010; Wang et al., 1998; Hekmatpour and Salehi, 2005; Rogin et al., 2008; DeOrio et al., 2009; Chang and Wang, 2010; Chung et al., 2011). GoldMine, for the first time developed a static analysis guided statistical analysis-based technique to generate hardware assertions (GoldMine, 2024; Hertz et al., 2013) in Linear Temporal Logic (Pnueli, 1977) in a fully automated way. While GoldMine and follow-up research (Hertz et al., 2019; Pal et al., 2020; Malburg et al., 2017; Reza Heidari Iman et al., 2024; Heidari Iman et al., 2021; Danese et al., 2017; Germiniani and Pravadelli, 2022a; Heidari Iman et al., 2022; Germiniani and Pravadelli, 2022b; Deutschbein et al., 2021; Witharana et al., 2023; Ayalasomayajula et al., 2024; Pal et al., 2020; Heidari Iman et al., 2022; Ghasempouri and Pravadelli, 2015) made assertions accessible beyond the hardware verification engineers, most of those methods still failed to scale to large industry-scale designs due to the algorithmic complexity of the underlying static analysis.

With recent advances in generative AI models, especially Large Language Models (LLMs), there is a renewed interest in the research community to harness the power of LLMs for assertion generation task. Most recent assertion generation approaches (Liu et al., 2023; Orenes-Vera et al., 2023; Kande et al., 2023; Fang et al., 2024; Mali et al., 2024; Pei et al., 2023) treat LLMs as ***black-box*** and use *prompt engineering* to iteratively refine the set of generated assertions. However, there is no in-depth study nor a dataset to evaluate how well different state-of-the-art (SOTA) LLMs perform on generating the correct set of assertions without designer developed prompts.

To address this gap, we propose AssertionBench: the first comprehensive benchmark to quantify the efficacy of SOTA LLMs for the assertion generation task. The benchmark contains 100 curated designs of varying complexity encompassing a broad spectrum of design types, including encoders, decoders, and arithmetic operations such as addition, multiplication and 2's complement in Floating Point Units, along with their formally verified assertions facilitating future research in exploring the applicability of LLMs in assertion generation. In this work we quantify the quality of the LLM-generated assertions prompted with a collection of labeled designs (and their formally verified assertions) as in-context learning (ICL) examples.

2 The AssertionBench Benchmark

AssertionBench[1] contains ICL example designs and and test designs from OpenCores (OpenCores, 2024). The benchmark contains combinational and sequential hardware designs containing up to 1250 lines of codes (LoCs) (Danial, 2021) excluding comments and blank lines.

In our benchmark, each ICL example consists of a Verilog design and its formally verified assertions, generated from GOLDMINE (Pal et al., 2020) and HARM (Germiniani and Pravadelli, 2022b), and verified using Cadence JasperGold (Cadence, 2024). AssertionBench primarily contains Verilog designs for benchmarking for the following reasons.

(a) The most recent work on LLM-assisted hardware designs focuses on Verilog, the predominant language for hardware design in industry and academia. For example, recent works such as VerilogEval (Liu et al., 2023), MG-Verilog (Zhang et al., 2024), Isadora (Deutschbein et al., 2021) and HARM (Germiniani and Pravadelli, 2022b) solely focus on Verilog. Our work aligns with the predominant and widely used hardware design language (HDL) in the state-of-the-art research and practice.

(b) To our knowledge, no public domain assertion generation tool is available to generate assertions for VHDL/SystemC designs. The only work that mines assertions from SystemC that we can find is Liu et al. (2011). However, we could not find the actual implementation of the tool in the public domain. Such assertions are necessary as ICL examples. This emphasizes the value of AssertionBench in complementing existing research and in future work we plan to augment it with VHDL/SystemC designs and corresponding assertions.

The key research question (RQ) that AssertionBench aims to address is ***whether we can effectively leverage state-of-the-art (SOTA) deep learning (DL) tools***, such as LLMs, to assist verification engineers in crafting assertions for large hardware designs and overcome the shortcomings of the classical static and dynamic analysis based techniques. There are ongoing efforts to leverage LLMs to alleviate the shortcomings. Our effort in designing AssertionBench is the first step towards ensuring that as we develop LLM-assisted techniques for assertion generation, we remain aware of the insights garnered by this work and avoid pitfalls. This benchmark and the framework will allow us to quantitatively and systematically compare the fitness of existing and future closed and open-sourced LLMs for the assertion generation task.

The ICL example set comprises fundamental sequential designs (aka designs with a clock) such as Arbiter, T Flip-Flop, and combinational designs (aka designs without a clock) such as Half Adder, Full Adder, and Full Subtractor. We consider the corresponding concurrent assertions (sequence subset of SVA) for ICL, and the which contain

[1] https://github.com/achieve-lab/assertion_data_for_LLM.

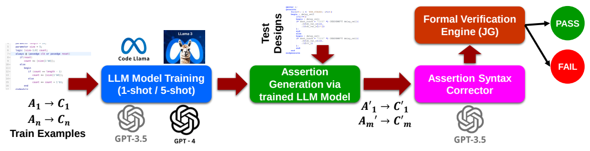

Figure 1: **Our evaluation framework**. **JG**: JasperGold Formal Property Verification Engine.

overlapped and non-overlapped implication operators (SystemVerilog, 2024). Our test design set consists of highly curated 100 Verilog designs (both sequential and combinational designs) of varying complexity that are up to 20× bigger than those in the ICL examples in term of LoCs, to evaluate LLMs' 1-shot and 5-shot assertion generation ability. We wanted to understand if LLMs can learn about an assertion and how it relates an assertion to the design source code using simple examples. ***Our main goal was to evaluate if the LLMs can transfer the learned knowledge successfully to much bigger designs***. A successful demonstration of such transfer of the learned knowledge would pave the path that LLMs can be an effective tool for assertion generation at scale, likely for industrial-scale designs even when learned and/or trained using smaller design source codes. We also wanted to understand ***if LLMs would excel or struggle for the restricted subset of assertion classes***. Our results (c.f., Section 4) show that LLMs struggle to generate correct assertions for this restricted subset. Hence it is futile to delve further into more complex assertion constructs unless we have clear insight into why LLMs are failing for the restricted subcases. In this effort, we develop those insights.

3 Experimental Setup

Figure 1 shows our evaluation framework. We evaluated four SOTA LLMs GPT-3.5 (Ye et al., 2023), GPT-4o (OpenAI et al., 2024), CodeLLaMa 2 (Rozière et al., 2024), and LLaMa3-70B (Meta, 2024) using the proposed AssertionBench on the task of predicting correct or valid assertions.

Compute Platform: We use UIUC NCSA's public Delta Cluster (NCSA, 2024) for our experiments. We use GPU nodes containing 1-way, 4-way, and 8-way NVIDIA A40 (with 48GB GDDR6) and A100 (with 40GB SXM) GPUs to perform k-shot learning and inference. Each 1-way and 4-way GPU computing node has 256 GB RAM, and each 8-way GPU computing node has 2 TB RAM.

Hyperparameters: We use default hyperparameters for all the LLM models under consideration. Specifically, the number of maximum output tokens has been set at 1024, employing a greedy decoding strategy and maintaining a *temperature* of 1.0, *top_p* of 0.95. The *random seed* has been configured to 50.

```
You are an expert in SystemVerilog
    Assertions.
Your task is to generate the list of
    assertions to the given verilog
    design. An example is shown below.
    Generate only the list of assertions
     for the test program with no
    additional text.
Program 1: module arb2(clk, rst, req1,
    req2, gnt1, gnt2); input clk, rst;
    ...
Assertions 1: (state == 1 & req2 == 1)
    |-> (gnt1 == 0);...
Test Program:
module fifo_mem #(parameter DEPTH=8,
    DATA_WIDTH=8, PTR_WIDTH=3) ( input
    wclk, w_en, rclk, r_en, input [
    PTR_WIDTH:0] b_wptr,  ...
Test Assertions:
```

Figure 2: **An example of the prompt for 1-shot learning**. The example consists of a tuple, a Verilog design (`Program 1`) and a set of formally verified assertions for the design (`Assertions 1`). The `Test Program` is the Verilog design for which we generate assertions using the trained LLM.

Pre-trained Models and Verification Tool: We use pre-trained LLMs from the HuggingFace model repository (HuggingFace, 2024) for AssertionBench evaluation. We have also used Python 3.11 and Cadence JasperGold (JG) version 2022.06p002 for formally verifying the assertions generated from the test Verilog designs. We use two SOTA classical tools GoldMine (Pal et al., 2020) and HARM (Germiniani and Pravadelli, 2022b) to generate *example assertions* from different Verilog designs in the IC set. Below, we summarize the four LLMs that we evaluate using AssertionBench.

• ***GPT-3.5*** is a commercial autoregressive LLM (from OpenAI) based on the GPT architecture, pretrained on extensive text corpora and fine-tuned for NLP tasks.

• ***GPT-4o*** is a unified multimodal transformer model processing text, vision, and audio, with enhanced reasoning and programming capabilities over previous iterations.

• ***CodeLLaMa 2*** is a suite of autoregressive transformer models for code and text generation, including a 70B parameter variant that uses Grouped-

Query Attention for scalable inference.

• ***LLaMa3-70B*** is a 70B-parameter transformer with an 8,192-token context window, pre-trained on 15 trillion tokens from publicly available datasets.

Evaluation Protocol: To evaluate effectiveness of the different LLMs, the few-shot testing regime consists of 1-shot and 5-shot ICL examples. Each example is a tuple consisting of a Verilog design source code and a set of formally verified assertions containing up to 10 assertions. Figure 2 shows our prompt consisting of four parts – (i) an English language description of the task, (ii) the Verilog design, (iii) an assertion in SystemVerilog Assertion (SVA) (SystemVerilog, 2024) format, and (iv) a test Verilog design. Next, we prompt each LLM with the ICL examples and evaluate them on 100 test Verilog designs to infer assertions. In our experiments, we have found all of the LLMs generate syntactically erroneous assertions, *i.e.*, each LLM fails to learn the SVA syntax from the ICL examples. Consequently, we use a syntax corrector (SC) using GPT-3.5 and feed the output of the SC to JG to evaluate the quality of the generated assertions. Any other SVA-compatible formal property verifier (FPV) will work as well.

Metrics: We evaluate the assertions generated for the test programs using the following three metrics for each LLM – (i) **Pass** quantifies the fraction of generated assertions that FPV attests as valid for the design; (ii) **Fail** quantifies the fraction of generated assertions that FPV attests as wrong with a counterexamples trace (CEX); and (iii) **Error** quantifies the fraction of generated assertions for which the FPV identifies syntactic errors even after syntax correction. We have not reported any other metric, *e.g.*, assertion coverage (Athavale et al., 2014), as that is meaningful only when an assertion is valid and one wants to quantify the quality of the assertion or would like to induce a ranking on assertions (Pal et al., 2020). In current work, we did not target to quantify the quality of the assertions neither did we want to induce a rank on them. Rather we focused on the ability of the SOTA LLMs on generating correct assertions.

4 Experimental Results

We show our overall experimental results in Figure 3. We make following observations.

Observation 1: **Most LLMs generate valid assertions with an increasing number of ICL examples** (c.f., Figure 3a-3d). GPT-3.5, GPT-4o, and CodeLLaMa 2 show on average an improvement of 2×, 1.2×, and 1.12× for valid assertion generation, respectively, when moved from 1-shot learning to 5-shot learning. However, LLaMa3-70B model loses accuracy from 31% to 24% on the same dataset. Our analysis shows in many cases, LLaMa3-70B either fails to generate assertions or generates syntactically wrong assertions or tries to generate codes in a new programming language (*e.g.*, Java). This experiment shows that *there is considerable scope for improving the LLaMa3-70B model for this task, likely via fine-tuning the pre-trained LLaMa3-70B model.*

Observation 2: **An enhanced model does not necessarily ensure a better semantic or syntactic understanding**. For GPT-3.5 (c.f., Figure 3a), with an increase in the number of ICL examples, the LLM was able to produce more syntactically correct assertions, however, majority of corrected assertions (on average up to 24%) generated a CEX when verified with JG. For GPT-4o, the results were more consistent in terms of syntactically correct and failing assertions from 1-shot and 5-shot learning (c.f., Figure 3b). For CodeLLaMa 2 and LLaMa3-70B, with increase in the number of ICL examples, the fraction of failed assertions decreased (on average up to 12% for CodeLLaMa 2 and LLaMa3-70B, c.f., Figure 3c and Figure 3d), however, both models generated more syntactically wrong assertions (on average up to 19% more for LLaMa3-70B). Our analysis shows that with 1-shot, the variation in types of assertions in the ICL examples were limited. Consequently, LLaMa3-70B learned the syntax. However, in 5-shot learning, we have more variations in assertion syntax which made LLaMa3-70B's learning task difficult. This experiment shows that *increasing the number of ICL examples will not necessarily improve LLM's consistency in generating passing, failing, and syntactically correct assertions*. Further, our analysis shows that *LLMs that are more performant on standard LLM benchmarks does not necessarily have a better semantic understanding when it comes to assertion generation.*

Observation 3: **GPT-4o is relatively more consistent and superior for assertion generation task** (c.f., Figure 3e-3f). Our experiment shows that GPT-4o generates on an average up to 15.6% more valid assertions as compared to other LLMs for both 1-shot and 5-shot learning. Additionally, GPT-4o produced fewer CEX generating as-

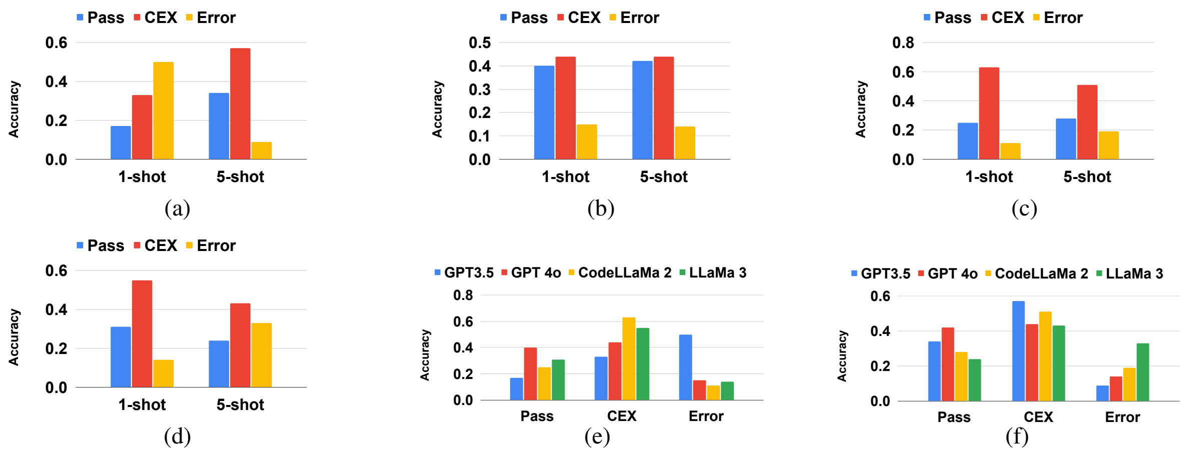

Figure 3: **Comparison of accuracy of generated assertions**. (a) Assertion accuracy comparison for GPT-3.5. (b) Assertion accuracy comparison for GPT-4o. (c) Assertion accuracy comparison for CodeLLaMa 2. (d) Assertion accuracy comparison for LLaMa3-70B. (e) $k = 1$-shot assertion accuracy. (f) $k = 5$-shot assertion accuracy. **CEX**: Counterexamples trace.

sertions and syntactically incorrect assertions as compared to other LLMs. This experiment shows that GPT-4o *is more beneficial for assertion generation as compared to the other LLMs*.

Observation 4: **All LLMs need considerable improvement for assertion generation task** (c.f., Figure 3). Our analysis shows that none of the LLM models can generate valid assertions an average of no more than 44% accuracy whereas up to 63% generated assertions produce CEX and on average up to 33% of generated assertions are syntactically wrong. Clearly, for LLMs to be of practical usage for any realistic industrial-scale design, considerable improvement needs to be made. Specifically, *the LLMs need to capture the semantic meaning of the Verilog designs for automatically producing a higher fraction of valid assertions without iterative human prompting*.

Remarks: In this work, we have refrained from reporting the coverage of assertions. We emphasize that unlike code-based coverage metrics, *e.g.*, line / statement coverage, branch coverage, condition coverage, FSM coverage, etc., there is no well-defined notion of assertion coverage. To the best of our knowledge, the only work that connects assertion's coverage of the design code is by Athavale et al. (2014). They defined correctness-based coverage of an assertion as identifying the design statements / codes that contribute to its non-vacuous satisfaction. However, such coverage, *i.e.*, assertion coverage as defined by Athavale et al. (2014), only makes sense when an assertion is correct (*i.e.*, valid) and one wants to quantify the quality of the correct assertion or would like to induce a ranking on assertions, *e.g.*, Pal et al. (2020); Ghasempouri and Pravadelli (2015). In current work, we did not target to quantify the quality of the generated assertions and neither did we want to induce a rank on generated assertions; rather we focused on the fitness of the current and future commercial and open-source LLMs on generating correct assertions. Such coverage (and ranking) would make perfect sense if we pursue the overarching goal of developing task-specific LLMs for assertion generation to quantify the quality of the LLM-generated assertions which in turn would quantify the quality of the task-specific LLMs.

5 Conclusion and Future Work

This work introduces AssertionBench to evaluate the current and future commercial and open-source LLMs for the assertion generation task. No prior work comprehensively benchmarks SOTA LLMs for assertion generation, especially for HDLs. To our knowledge, AssertionBench is the first such benchmark to quantitatively compare various LLMs in terms of goodness for the task of assertion generation. Although there is no LLM that consistently outperforms other LLMs, we notice several promising trends and research directions to enhance the practical applicability of LLMs for assertion generation task, which will further accelerate SoC and hardware design verification. As LLM research is growing at a tremendous pace both in commercial and academic research, we plan to maintain the benchmark and augment its learning and test set with more complex designs and their formally verified assertions to further stress test models.

6 Limitations

We identify the following limitations of this work in terms of the dataset and the evaluation methodology.

- **Dataset**: In the scope of this study, our primary focus is on Verilog designs, given its status as the predominant hardware design language. Moving forward, it will be intriguing to develop benchmarks for assertions in alternative hardware languages, *e.g.*, VHDL, SystemVerilog, and SystemC, thereby expanding the scope of our analysis to encompass a broader range of design paradigms. Additionally, AssertionBench considers only a few temporal assertions with shallow temporality. It would be interesting to increase the temporal depth to capture design behaviors that cut across multiple clock cycles and evaluate LLM's ability to learn and generate assertions to capture such behaviors succinctly.

- **Assertion Objective**: In this work, we primarily focused on the assertions that capture design functionality. It would be interesting to enhance and augment AssertionBench with security assertions to evaluate the LLM's ability to capture and summarize security violations/concerns from hardware design source code.

- **Quantitative Assertion Ranking**: In this work, we primarily focused on correctness of an assertions without quantifying and ranking the subtlety of the captured design behavior (Pal et al., 2020). It would be interesting to include such rankings in the ICL examples and evaluating LLM's capability to automatically rank generated assertions to quantify captured design behavior.

- **Modeling**: In this paper, we assessed the few-shot assertion generation capabilities of SOTA language models. In future work, it will be interesting to fine-tune language models for assertion generation and evaluate their performance on AssertionBench.

- **Evaluation**: In future work, it will be valuable to conduct a more detailed evaluation of model errors to better understand the specific limitations of each LLM for assertion generation.

References

Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. 2014. Code Coverage of Assertions Using RTL Source Code Analysis. *Design Automation Conf. (DAC)*.

Avinash Ayalasomayajula, Nusrat Farzana, Debjit Pal, and Farimah Farahmandi. 2024. Prioritizing Information Flow Violations: Generation of Ranked Security Assertions for Hardware Designs. *IEEE Int'l Symp. on Hardware Oriented Security and Trust (HOST)*.

Cadence. 2024. JasperGold Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html. Accessed: February 7, 2025.

Po-Hsien Chang and Li.-C Wang. 2010. Automatic Assertion Extraction via Sequential Data Mining of Simulation Traces. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*.

Chih-Neng Chung, Chia-Wei Chang, Kai-Hui Chang, and Sy-Yen Kuo. 2011. Applying Verification Intention for Design Customization via Property Mining Under Constrained Testbenches. *Int'l Conf. on Computer Design: VLSI in Computers and Processors, (ICCD)*.

Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. 2017. A-TEAM: Automatic Template-based Assertion Miner. *Design Automation Conf. (DAC)*.

Albert Danial. 2021. CLOC. https://github.com/AlDanial/cloc. Accessed: February 7, 2025.

Andrew DeOrio, Adam B. Bauserman, Valeria Bertacco, and Beth C. Isaksen. 2009. Inferno: Streamlining Verification With Inferred Semantics. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. 2021. Isadora: Automated Information Flow Property Generation for Hardware Designs. *Workshop on Attacks and Solutions in Hardware Security (ASHES)*.

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly Detecting Relevant Program Invariants. *Int'l Conf. on Software Engineering (ICSE)*.

Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. 2024. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *arXiv*.

Samuele Germiniani and Graziano Pravadelli. 2022a. Exploiting Clustering and Decision-Tree Algorithms to Mine LTL Assertions Containing Non-boolean Expressions. *IFIP/IEEE Int'l Conf. on Very Large Scale Integration (VLSI-SoC)*.

Samuele Germiniani and Graziano Pravadelli. 2022b. HARM: A Hint-Based Assertion Miner. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Tara Ghasempouri and Graziano Pravadelli. 2015. On The Estimation of Assertion Interestingness. *IFIP/IEEE Int'l Conf. on Very Large Scale Integration (VLSI-SoC)*.

GoldMine. 2024. GoldMine: An Automatic Assertion Generation Tool. http://goldmine.csl.illinois.edu/. Accessed: February 7, 2025.

Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. 2005. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. *Design Automation Conf. (DAC)*.

Mohammad Reza Heidari Iman, Jaan Raik, Maksim Jenihhin, Gert Jervan, and Tara Ghasempouri. 2021. A Methodology for Automated Mining of Compact and Accurate Assertion Sets. *IEEE Nordic Circuits and Systems Conference (NorCAS)*.

Mohammad Reza Heidari Iman, Jaan Raik, Gert Jervan, and Tara Ghasempouri. 2022. IMMizer: An Innovative Cost-Effective Method for Minimizing Assertion Sets. *Euromicro Conference on Digital System Design (DSD)*.

A. Hekmatpour and A. Salehi. 2005. Block-based Schema-driven Assertion Generation for Functional Verification. *Asian Test Symposium (ATS)*.

Samuel Hertz, Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. 2019. A Figure of Merit for Assertions in Verification. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*.

Samuel Hertz, David Sheridan, and Shobha Vasudevan. 2013. Mining Hardware Assertions With Guidance From Static Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

HuggingFace. 2024. Model Repository. https://huggingface.co/. Accessed: February 7, 2025.

Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. 2023. LLM-assisted Generation of Hardware Assertions. *arXiv*.

Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. 2011. Automatic Generation of Assertions from System Level Design Using Data Mining. *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *arXiv*.

Jan Malburg, Tino Flenker, and Görschwin Fey. 2017. Property Mining Using Dynamic Dependency Graphs. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*.

Bhabesh Mali, Karthik Maddala, Sweeya Reddy, Vatsal Gupta, Chandan Karfa, and Ramesh Karri. 2024. ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation. *arXiv*.

Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/. Accessed: February 7, 2025.

NCSA. 2024. NCSA Delta. https://www.ncsa.illinois.edu/research/project-highlights/delta/. Accessed: February 7, 2025.

OpenAI, Josh Achiam, and et al. 2024. GPT-4 Technical Report. *arXiv*.

OpenCores. 2024. https://opencores.org/. Accessed: February 7, 2025.

Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using LLMs to Facilitate Formal Verification of RTL. *arXiv*.

Debjit Pal, Spencer Offenberger, and Shobha Vasudevan. 2020. Assertion Ranking Using RTL Source Code Analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Corina S. Păsăreanu and Willem Visser. 2004. Verification of Java Programs Using Symbolic Execution and Invariant Generation. *Int'l SPIN Workshop on Model Checking of Software (SPIN)*.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason About Program Invariants? *Int'l Conf. on Machine Learning (ICML)*.

G. Pinter and I. Majzik. 2005. Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements. *Int'l Symp. on High-Assurance Systems Engineering (HASE)*.

Amir Pnueli. 1977. The Temporal Logic of Programs. *Annual Symp. on Foundations of Computer Science (SFCS)*.

Mohammad Reza Heidari Iman, Gert Jervan, and Tara Ghasempouri. 2024. ARTmine: Automatic Association Rule Mining with Temporal Behavior for Hardware Verification. *Design, Automation, and Test in Europe (DATE)*.

Frank Rogin, Thomas Klotz, Gorschwin Fey, Rolf Drechsler, and Steffen Rulke. 2008. Automatic Generation of Complex Properties for Hardware Designs. *Design, Automation, and Test in Europe (DATE)*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. *arXiv*.

SystemVerilog. 2024. 1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. https://ieeexplore.ieee.org/document/8299595. Accessed: February 7, 2025.

A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. 2001. A Technique for Invariant Generation. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

L.-C. Wang, M.S. Abadir, and N. Krishnamurthy. 1998. Automatic Generation of Assertions for Formal Verification of PowerPC/sup TM /microprocessor Arrays Using Symbolic Trajectory Evaluation. *Design Automation Conf. (DAC)*.

Hasini Witharana, Aruna Jayasena, Andrew Whigham, and Prabhat Mishra. 2023. Automated Generation of Security Assertions for RTL Models. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*.

Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. *ACM Comput. Surv. (CS)*.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. *arXiv*.

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Lin. 2024. MG-Verilog: Multi-grained dataset towards enhanced llm-assisted verilog generation. *IEEE Int'l Workshop on LLM-Aided Design (LAD)*.