Are LLMs Ready for Practical Adoption for Assertion Generation?

Vaishnavi Pulavarthi, Deeksha Nandal
Electrical and Computer Engg. Dept.
University of Illinois Chicago
Chicago, USA
{vpulav2, dnanda6}@uic.edu

Soham Dan
Microsoft
sohamdan@microsoft.com

Debjit Pal
Electrical and Computer Engg. Dept.
University of Illinois Chicago
Chicago, USA
dpal2@uic.edu

***Abstract*—Assertions have been the de facto collateral for simulation-based and formal verification of hardware designs for over a decade. The quality of hardware verification, *i.e.*, detection and diagnosis of corner-case design bugs, is critically dependent on the quality of the assertions. With the onset of generative AI such as Transformers and Large-Language Models (LLMs), there has been a renewed interest in developing novel, effective, and scalable techniques of generating functional and security assertions from design source code. While there have been recent works that use commercial-of-the-shelf (COTS) LLMs for assertion generation, there is no comprehensive study in quantifying the effectiveness of LLMs in generating syntactically and semantically correct assertions. In this paper, we first discuss AssertionBench from our prior work, a comprehensive set of designs and assertions to quantify the goodness of a broad spectrum of COTS LLMs for the task of assertion generations from hardware design source code. Our key insight was that COTS LLMs are not yet ready for prime-time adoption for assertion generation as they generate a considerable fraction of syntactically and semantically incorrect assertions. Motivated by the insight, we propose AssertionLLM, a first of its kind LLM model, specifically fine-tuned for assertion generation. Our initial experimental results show that AssertionLLM considerably improves the semantic and syntactic correctness of the generated assertions over COTS LLMs.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

System-on-Chip (SoC) designs are crucial for many safety-critical computing applications, including vehicular systems, military, and industrial automation. SoCs often use sensitive and users' private data to perform numerous complex computations. It is crucial for our national and personal well-being to ensure that the SoCs are functionally correct, safe, and secure.

Assertions are mathematical encoding of desired design properties that should hold True for a design. Assertions are widely used for hardware design validation throughout its life cycle, *e.g.*, pre-silicon formal verification and simulation-based verification, emulation, and often synthesized in a fabricated chip for post-silicon validation and in-field debug and diagnosis. In the past decade, assertion-based Verification (ABV) [1] has emerged as the de facto standard to verify the security and functional correctness of hardware designs. However, crafting a succinct yet expressive set of assertions that capture subtle and important hardware design behaviors is a tedious and time-consuming task, requiring a considerable amount of human ingenuity. Too many assertions (i) can negatively affect verification performance with a prolonged verification closure and (ii) may require a substantial amount of on-chip resources, whereas too few assertions may result in insufficient design coverage causing corner case design bugs to escape to production and mass manufacturing. The ever increasing hardware design complexity and rapidly broadening target applications (*e.g.*, deep learning, AI) have only worsened the problem. ***Consequently, developing an automated and scalable technique is crucial to rapidly generate a succinct set of hardware design properties targeting design functionality and security***.

A considerable amount of research has leveraged two different paradigms – lightweight static analysis of design source code and formal verification [2]–[4], and data-driven statistical analysis, *e.g.*, data mining [5]–[10]. While static analysis can generalize and capture corner-case design behaviors, it suffers from prohibitive computational complexity limiting its scalability to industrial-scale designs. Alternatively, data-driven dynamic analysis can scale to large designs with a considerable amount of trace data due to its computational efficiency, however, it often generates spurious design properties due to the lack of design insights and domain context. More recently, researchers have proposed assertion generation techniques that combine static analysis and data-driven dynamic analysis [11]–[13] and developed algorithms to induce ranking on such automatically generated assertions based on the subtle design behavior they capture [14]. However, all such techniques generate a large number of assertions, many of which are redundant and do not capture model-level or system-level design behaviors, and fail to scale to large industry-scale designs due to the algorithmic complexity of the underlying static analysis. Despite intense research across academia and industry over the last decade, there is a widening gap between assertion solutions and the industry's actual requirements in terms of assertion quality for complex hardware designs.

With recent advances in deep-learning (DL) and generative AI models, especially Large-Language Models (LLMs), *e.g.*, GPT-3.5, GPT-4o, CodeLLaMa 2, and LLaMa3-70B, there is a renewed interest to harness the power of LLMs to tame the ever-widening gap. Most recent assertion generation approaches (c.f., Section VIII) treat a LLM model as a ***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

978-3-9826741-0-0/DATE25/© 2025 EDAA

```
1  module arb2(clk, rst, req1, req2, gnt1, gnt2);
2  input clk, rst, req1, re2;
3  output gnt1, gnt2;
4  reg gnt_, gnt1, gnt2;
5  always @(posedge clk or posedge rst)
6    if(rst)
7       gnt_ <= 0;
8    else
9       gnt_ <= gnt1;
10 always @(*)
11   if (gnt_)
12   begin
13      gnt1 = req1 & req2;
14      gnt2 = req2;
15   end
16   else
17   begin
18     gnt1 = req1;
19     gnt2 = req2 & ~req1;
20   end
21 endmodule
```

Fig. 1: **A Verilog code for a 2-port Arbiter** [14].

		Pre-condition	
		Covered	**Unreachable**
Post-Condition	**True**	**Valid**	**Vacuous**
	False	**CEX**	

Fig. 2: **Assertion status based on pre-condition and post-condition evaluation**. **CEX**: Counter example.

evaluate the fit of different state-of-the-art (SOTA) LLM models for generating a succinct and correct set of assertions without a considerable amount of designer-developed prompts.

In this work, first, we discuss our prior work AssertionBench [15], the first comprehensive benchmark consisting of 100 curated hardware designs of varying complexity and their formally verified assertions to quantify the efficacy of SOTA and upcoming LLMs for assertion generation. Our primary focus is to quantify the quality of the generated assertions from SOTA LLMs learned on a set of labeled designs and their formally verified assertions. Our ***key insight*** is that almost all SOTA LLMs generate a considerable fraction of syntactically and semantically incorrect assertions. ***Leveraging this insight, we develop AssertionLLM, a fine-tuned LLM model that can automatically generate substantially higher fraction (up to 25%) of syntactically and semantically correct assertions from design source codes without any iterative inputs from the verification engineer***. We further outline several research ***challenges*** and ***opportunities*** that are worth pursuing to truly exploit the potential of generative AI for assertion generation.

II. Background

A. Assertions: Syntax, Semantics, and Validity

We consider a hardware design $\mathcal{D}$ in Verilog[1] as a composition of a set of concurrent processes P_i, *e.g.*, (`always` and `assign` blocks). Let V be the set of design signals, $\mathcal{I} \subset V$ be the set of input signals, $\mathcal{O} \subset V$ be the set of output signals, and $\mathcal{R} \subset V$ be the set of registers. Figure 1 shows a Verilog design $\mathcal{D}$ of a 2-port Arbiter, consisting of two concurrent processes P_1 (line 6) and P_2 (line 11), and $V = \{clk, rst, req1, req2, gnt1, gnt2, gnt\_\}$, $\mathcal{I} = \{clk, rst, req1, req2\}$, $\mathcal{O} = \{gnt1, gnt2\}$, and $\mathcal{R} = \{gnt\_\}$.

An assertion is a temporal formula in LTL [16] of the format $P = \mathcal{G}(A \rightarrow C)$ where the antecedent A is of the form $A = \bigwedge_{i=0}^{m} \mathcal{X}^i(A^i)$ and consequent C is of the form $C = \mathcal{X}^n(C^n)$, where $n \geq m$. Each A_i (C_i) is a proposition and is a (var, val) pair where $var \in V$ and $val \in \{0, 1\}$. $\mathcal{X}$ is called the next-cycle operator and $\mathcal{X}^i(i \geq 0)$ is equal to a delay of i clock cycles $\underbrace{\mathcal{X}\mathcal{X}\ldots\mathcal{X}}_{i \text{ times}}$. Although SystemVerilog Assertion (SVA) [17] defines a rich set of grammar for assertions, we consider a restricted subset (sequential assertion) as captured by P. We say an assertion P is **True** (Valid) if $\mathcal{D} \models P$ (read as $\mathcal{D}$ models P), otherwise, the assertion is **False**, *i.e.*, $\mathcal{D} \not\models P$ and there exists a Boolean value assignment to a subset of design signals known as **CEX** (counter-example) that shows a refutation of the assertion P on $\mathcal{D}$. The implication operator $\rightarrow$ in P are of two types, ***overlapped*** and ***non-overlapped***. The *overlapped* implication operator ($\rightarrow$) implies *if there is a match on the antecedent A, then the consequent C is evaluated in the same clock cycle* whereas the *non-overlapped* implication operator ($\Rightarrow$) implies *if there is a match on the antecedent A, then the consequent C is evaluated in the next clock cycle*. In Figure 2, we show the assertion evaluation status. Note $A \rightarrow C$ can be re-written as $\neg A \vee C$. Consequently, if pre-condtion A is **unreachable** (or False), then the assertion P is vacuously True (*i.e.*, $\neg False \vee C$). If pre-condition A is True and post-condition C is True as well, the assertion is reported to be Valid (*i.e.*, $\mathcal{D} \models P$), otherwise, if the post-condition C is False, then a counter example **CEX** is generated.

For the Arbiter of Figure 1, consider assertions $P1$: $\mathcal{G}((req1 == 1 \wedge req2 == 0) \rightarrow (gnt1 == 1))$ and $P2$: $\mathcal{G}((req2 == 0 \wedge gnt\_ == 1) \wedge \mathcal{X}(req1 == 1) \rightarrow \mathcal{X}(gnt1 == 1))$. The assertion $P1$ evaluates **True** if $req1$ is `1'b1` and $req2$ is `1'b0` at the current clock cycle, then $gnt1$ is `1'b1` in the current clock cycle. The assertion $P2$ evaluates **True** if $req2$ is `1'b0` and $gnt\_$ is `1'b1` in the current cycle, $req1$ is `1'b1` in the next cycle, then $gnt1$ is `1'b1` in the cycle after (*i.e.*, in the 2^{nd} cycle). Note that $P2$ can be re-written using the *non-overlapped* implication operator, $P2$: $\mathcal{G}((req2 == 0 \wedge gnt\_ == 1) \wedge \mathcal{X}(req1 == 1) \Rightarrow (gnt1 == 1))$ where the $\Rightarrow$ subsumes the $\mathcal{X}$ operator in the consequent. On discharging a proof for $P1$ and $P2$ using a formal property verification (FPV) engine[2], we find $P1$ is a **valid** assertion whereas $P2$ generates a **CEX**.

B. Large-Language Models

Large-Language Models (LLMs) are an instance of generative AI built on top of encoder-decoder transformer architectures [19]. LLMs can be classified primarily in three classes, (i) encoder-only LLMs [20], (ii) decoder-only LLMs [21], and (iii) encoder-decoder LLMs [22]. Encoder-only LLMs

[1]We consider Verilog as the demonstration vehicle for this work, however, our work can naturally be extended to other hardware design languages, *e.g.*, VHDL, SystemC, and other hardware description languages.

[2]We use Cadence JasperGold [18], however, any other FPV tool will work.

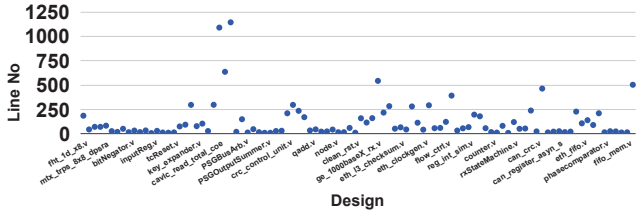


Fig. 3: **Design details in the test set in terms of the number of lines of code (excluding comments and blank lines).**

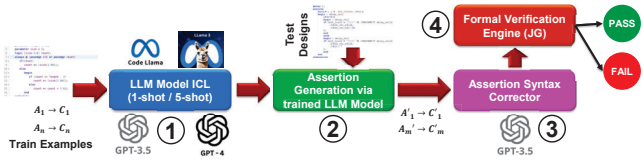


Fig. 4: **Framework to evaluate LLMs for assertion generation** [15]. **JG**: JasperGold Formal Property Verification Engine.

employ a bi-directional transformer during pre-training for each token to attend every other token, decoder-only LLMs employ unidirectional language modeling for each token to attend its predecessor tokens, and where tokens can only participate in previous tokens, and encoder-decoder LLMs employ denoising sequence-to-sequence pre-training objectives. The decoder-only LLM performs excellently in auto-regressive tasks such as code completion and generation. Since assertion generation is a special kind of code generation, in this work, we employ decoder-only LLMs, *e.g.*, GPT-3.5, GPT-4o, etc.

There are two distinct paradigms for LLM usage for different tasks – (i) in-context learning (ICL), where a foundational LLM (*e.g.*, GPT-4o) is seeded with a few examples of the desired task followed by deployment and (ii) fine tuning where a foundational LLM is trained with a small amount of high-quality downstream task-specific data to construct a task-specific LLM. In this work, we use ICL to evaluate the fitness of COTS LLMs for assertion generation and use finetuning to develop specialized LLMs for assertion generation tasks.

III. AssertionBench: Benchmark to Quantify Goodness of LLMs for Assertion Generation

AssertionBench[3] is a comprehensive suite of Verilog design and associated formally verified assertions to evaluate the goodness of the COTS LLMs for assertion generation [15]. AssertionBench consists of designs from OpenCores [23]. Figure 3 and Table I show representative details of the designs. Our benchmark consists of five ICL examples for 1-shot and 5-shot learning, where each example is a tuple consisting of a Verilog design and its formally verified assertions, generated from GoldMine [11] and HARM [13], and verified using Cadence JasperGold [18]. The training set comprises fundamental designs such as Arbiter, Half Adder, Full Adder, T-flip-flop, and Full Subtractor. Among these designs, Arbiter and T flip-flops are sequential, while the others are combinational. Our training set assertions contain both overlapped and non-overlapped implication operators. The test design set contains

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

```
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:
```

Fig. 5: **An example of the prompt for 1-shot learning** [15]. 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.

100 Verilog designs (split in combinational and sequential designs) from OpenCores [23] that are more complex than those in the training set, to evaluate LLMs' 1-shot and 5-shot learning. The set cover a wide variety of hardware including communication controllers, random number generators (RNG) for security hardware, Floating Point Unit (FPU), state machines, and flow control hardware. The test designs code size varies from 10 lines to 1150 lines (excluding blanks and comments) as measured by cloc [24].

IV. Experimental Setup

Evaluation Protocol: Figure 4 shows our evaluation framework. To evaluate the effectiveness of the different LLMs, our k-shot ICL consists of 1-shot and 5-shot in-context examples (ICE) (① in Figure 4). Each ICE is a tuple $\langle \mathcal{D}, \mathcal{A} \rangle$, where $\mathcal{D}$ is a Verilog source code and $\mathcal{A}$ is a set of formally verified assertions containing a minimum of two (2) and a maximum of 10 assertions with an average of 4.8 assertions per source code. We use a prompt as shown in Figure 5 consisting of four parts – (i) an English language description of the task in hand, (ii) an example Verilog design with newlines and comments removed, (iii) an example assertion in SVA format, and (iv) a test Verilog design with new lines and comments removed. Followed by training, we provide each trained model with 100 test Verilog programs to infer assertions (② in Figure 4). In our experiments, we have found all of the LLM models generate syntactically erroneous assertions, *i.e.*, each LLM fails to learn the SVA syntax from the training examples. Consequently, we use a syntax corrector (③ in Figure 4) using GPT-3.5 and feed the output of the syntax corrector to Cadence JasperGold FPV engine to evaluate the quality of the generated assertions. Note any other FPV engine compatible with SVA will work as well.

ICL Compute Platform: We use UIUC (University of Illinois Urbana-Champaign) NCSA's (National Center for Supercomputing Applications) Delta Cluster [25] to run 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.

TABLE I: **Details of a few representative designs in the test set of AssertionBench benchmark**.

Verilog Design	# of Lines	Design Type	Design Functionality
ca_prng	1144	Sequential	A compact Pattern Generator
cavlc_read_total_coeffs	1090	Sequential	Video Encoder for generic audio visual.
cavlc_read_total_zeros	637	Combinational	Video Encoder for generic audio visual.
ge_1000baseX_rx	544	Sequential	Verilog implementation of Physical Coding Sublayer (PCS) type.
MAC_tx_Ctrl	504	Sequential	An Ethernet MAC controller.

Pre-trained Models and EDA Tools: We use pre-trained LLMs from the HuggingFace [26] for evaluation and Cadence JasperGold version 2022.06p002 to formally verify the assertions generated from the test Verilog designs. We use two SOTA tools GOLDMINE [11], [14] and HARM [13] to generate assertions for Verilog designs in the ICE. Below, we summarize the COTS LLMs that we evaluate using AssertionBench.

1) **GPT-3.5** is a commercial LLM built using the GPT architecture [27]. It is part of OpenAI's GPT (Generative Pre-trained Transformer) series of models designed to understand and generate text based on the input it receives.

2) **GPT-4o** ('o' for "omni") is the newest model of OpenAI's GPT, which accepts any combination of input audio, image, video, and text and responds with an output consisting of image, audio, video, and text [21]. With larger training data, increased model size, and faster response than other GPT models, GPT-4o is a unified model for text, vision, and audio.

3) **CodeLLaMa 2** is a collection of generative text models developed by Meta [28] with parameters ranging from 7B to 70B. The model accepts only text as input and output. It is an auto-regressive language model. The context window length for CodeLLaMa 2 is 4096. The large 70B model uses Grouped-Query Attention for improved inference scalability.

4) **LLaMa3-70B** is available in two parameter sizes – 8B and 70B. The context window length for LLaMa3-70B is 8192 and is pre-trained with 15 Trillion tokens of publicly available data [29]. LLaMa3-70B excels at translation, contextual understanding, and dialogue generation. It has enhanced capabilities such as code generation, reasoning, and following instructions.

ICL Hyperparameters: For all LLMs under consideration, the hyperparameters have been set to their default values. Specifically, the maximum output tokens is set to 1024, employing a greedy decoding strategy and maintaining a *temperature* of 1.0 (most creative), *top_p* of 0.95. The *random seed* is set to 50.

Metrics: We evaluate the generated assertions from the test designs using following metrics for each k-shot ICL per LLM.

1) **Pass** quantifies the fraction of generated assertions that FPV engine attests as valid for the design. This includes the Vacuous and the Pass cases from Figure 2.

2) **Fail** quantifies the fraction of generated assertions that FPV engine attests as wrong for the design and generates a counterexample trace. This includes the Fail case from Figure 2.

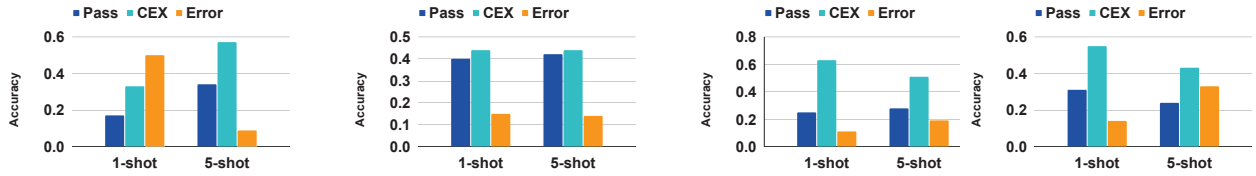
3) **Error**: It quantifies the fraction of generated assertions for which the FPV engine identifies one or more syntactic errors in the assertions even after syntax correction by the GPT-3.5.

V. OBSERVATIONS AND INSIGHTS FROM ASSERTIONBENCH

We depict our observations and insights [15] in Figure 6 and Figure 7 and discuss them below.

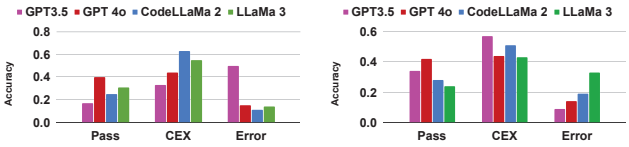
Observation 1: **Most LLMs generate valid assertions with an increasing number of ICL examples**. For the assertion generation task, all LLMs progressively generate more valid assertions when the number of ICL examples is increased as seen in Figure 6. GPT-3.5, GPT-4o, and CodeLLaMa 2 show on average 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, the LLaMa3-70B model loses accuracy from 31% to 24% on the same dataset. Our in-depth analysis shows in many cases, LLaMa3-70B either fails to generate assertions or generates syntactically wrong assertions (which even a syntax corrector fails to correct) or tries to generate codes in a new programming language (such as Java). ***This experiment shows that there is a 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 LLM does not necessarily ensure a better semantic or syntactic understanding**. In Figure 6, we do not see a correlation between the sophistication (in terms of the number of model parameters) of the LLMs and their ability to generate good assertions. For GPT-3.5 (c.f., Figure 6a), with an increase in the number of ICL examples, the LLM was able to produce more syntactically correct assertions, however, after such corrections, the majority of assertions (on average up to 24%) generated a CEX when verified with JasperGold. For GPT-4o, the results were more consistent in terms of syntactically (in)correct assertions for both 1-shot and 5-shot learning (c.f., Figure 6b). For CodeLLaMa 2 and LLaMa3-70B, with an increase in the number of ICL examples, the number of failed assertions decreased (on average up to 12% for CodeLLaMa 2 and LLaMa3-70B, c.f., Figure 6c and Figure 6d), however, both models generated more syntactically wrong assertions (on average up to 19% more for LLaMa3-70B). This observation is perplexing as one would expect with more number of parameters, LLaMa3-70B would be able to learn better to predict syntactically correct assertions. Our in-depth analysis shows that with a 1-shot, the variations in types of assertions in examples were limited. Consequently, LLaMa3-70B learned the assertion syntax. However, in 5-shot learning, we have more variations in the which made LLaMa3-70B's learning task difficult. ***This experiment suggests that increasing the ICL examples alone will not neces-***



(a) (b) (c) (d)

Fig. 6: **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. (a) $k = 1$-shot assertion accuracy. (b) $k = 5$-shot assertion accuracy. **CEX**: Counter Example trace.



(a) 1-shot. (b) 5-shot.

Fig. 7: **Comparison of accuracy of generated assertions in terms of passing, failing, (generating a counter example), and syntactically wrong assertions between different LLMs per k-shot learning where $k = 1$ and $k = 5$.**

sarily improve LLM's consistency in generating syntactically and semantically correct assertions.

Observation 3: **GPT-4o is relatively more consistent for assertion generation task**. In Figure 7, we compare different LLMs in terms of generating valid assertions for 1-shot and 5-shot learning. Our experiments show that GPT-4o is consistently superior in generating valid assertions for both 1-shot and 5-shot learning and generates, on average, up to 15.6% more valid assertions as compared to other LLMs. This trend remains valid with respect to assertions generating CEX and syntactically wrong assertions, *i.e.*, GPT-4o produced less CEX generating assertions and syntactically incorrect assertions as compared to other LLMs. ***This experiment shows that GPT-4o performs relatively better than other LLMs***.

Observation 4: **All LLMs need considerable improvement for assertion generation task**: In-depth analysis of Figure 6 and Figure 7 show that none of the LLMs can generate valid assertions with an average of no more than 44% accuracy whereas up to 63% generated assertions produces 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 hardware description languages, *e.g.*, Verilog, for generating higher fraction of valid assertions automatically without iterative human prompting. Our prior work [11], [14] shows that such critical insights are not directly available from the raw design source code and need auxiliary artifacts, such as Control-Data Flow Graph (CDFG), Variable Dependency Graph (VDG), Cone of Influence (COI), etc. ***Future research in applying LLMs for assertion generation should consider such auxiliary artifacts to develop assertion-specific LLMs***.

Evaluation of the four COTS LLMs using AssertionBench shows that no LLM consistently outperforms other LLMs.

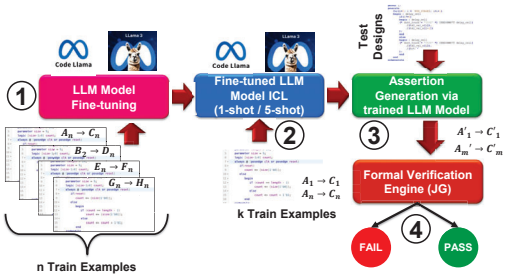


Fig. 8: **Our evaluation framework for AssertionLLM**.

Our analysis emphasizes that there is a considerable scope to enhance LLMs for assertion generation. ***There are two different ways*** – (i) enhance ICL with diverse ICL examples or (ii) develop an LLM specifically for assertion generation. Using this insight, we develop AssertionLLM as detailed in Section VI.

VI. AssertionLLM: LLM for Assertion Generation

AssertionLLM is an LLM specifically developed for the task of assertion generation. This is inline with the recent findings in other domains, *e.g.*, financial analysis, where a task-specific LLM has excelled for downstream tasks as compared to a foundational LLM. We use two different LLM foundational models – (i) CodeLLaMa 2 and (ii) LLaMa3-70B and finetune each of them using a large amount of data where each data point consists of a Verilog design and its formally verified assertions.

Figure 8 shows the end-to-end flow to finetune LLM for assertion generation and how we use the finetuned LLM to evaluate its goodness for assertion generation. Compared to Figure 4, we have removed the syntax corrector block (③ in Figure 4), and instead of using foundational LLM (① in Figure 4), we are using the fined-tuned LLM model for ICL (② and ③ in Figure 8).

We use the same compute platform and and hyperparameters as detailed in Section IV for finetuning. Additionally, we use 20 epochs to finetune each foundational LLM. We split the AssertionBench and use 75% of data for training and the remaining 25% for testing the goodness of the tuned LLM.

VII. Experimental Results on AssertionLLM

Figure 9 shows our experimental results on assertion generation task using finetuned CodeLLaMa 2 and LLaMa3-70B. We compare these results to that of results in Figure 6.

Observation 5: **Finetuning LLMs considerably improves fraction of correct assertions**: We observe that the finetuned

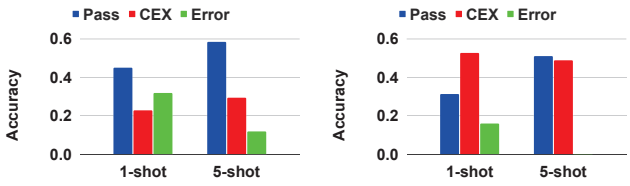
CodeLLaMa 2 increased proven assertions by 29% and 38% for 1-shot and 5-shot ICL, decreased assertions generating CEX by 48% and 33% for 1-shot and 5-shot ICL, respectively (c.f., Figure 6c and Figure 9a). With respect to LLaMa3-70B, fine tuning has increased proven assertions by 24% for 5-shot learning (c.f., Figure 6d and Figure 9b). However, for 1-shot ICL, the fraction of proven assertions has decreased by 4.7% and has increased the fraction of assertions generating CEX by 5.4% and 12% for 1-shot and 5-shot ICL, respectively. Further analysis shows that as the foundational CodeLLaMa 2 was trained on large corpora of codes, it learned assertion syntax and semantics better during finetuning. In contrast, foundational LLaMa3-70B training on general text corpora struggles to learn assertion syntax and semantics during finetuning. ***This experiment shows that it is crucial to select appropriate foundational LLM and dataset for an effective fine-tuned LLM for assertion generation***.

Observation 6: **Fine-tuning LLMs does not necessarily guarantee syntactic error-free assertions**: Fine-tuning LLMs does not necessarily nullify the fraction of syntactically erroneous assertions. Figure 9 shows that both finetuned CodeLLaMa 2 and LLaMa3-70B generate a considerable fraction (upto 38%) of syntactically erroneous assertions. ***In order to reduce the fraction of erroneous assertions, we speculate that we will require a more comprehensive dataset with a sufficient number of examples for the diverse syntax of assertions***.

VIII. Related Work

Automatic generation of assertions in hardware has been an active area of research for the past decade. IODINE is one of the earliest works for hardware assertion generation by analyzing dynamic program behavior with respect to a set of standard property templates [5]. Prior works have used static analysis [6], [7], dynamic simulation execution data [8]–[10], and data-driven statistical analysis guided by the lightweight static analysis of design source code [11], [30] for assertion generation. Following GoldMine, researchers have developed a wide variety of assertion generation techniques targeting hardware design functionality [12]–[14], [31]–[33] and hardware design security [34], [35], and to evaluate the quality of numerous assertions that automatic methods generate to aid the downstream verification tasks [14], [33], [36]. ***However, all these works suffer from following shortcomings*** – they (i) do not scale for industrial-scale designs, (ii) require a massive amount of trace data to generate assertions, (iii) generate numerous redundant design properties without any explanation on their usability for downstream verification tasks, (iv) fail to generalize the properties beyond what is seen in the trace, and (v) encompass an extremely small subset of SVA, limiting the expressibility and richness of the generated assertions.

Recently, massive success of LLMs, *e.g.*, GPT [21], LLaMa [37], Gemini [38], in diverse scientific, engineering, and medical applications have led researchers to investigate application of LLMs for hardware property generation [39]–[44]. ***However, almost all recent works on property generation using LLMs suffer from the following shortcomings*** – they (i) require considerable human efforts and deep understanding of the target hardware designs to devise *hand-crafted prompts* to generate and refine hardware properties, (ii) do not generalize the assertions, and (iii) do not consider execution traces, risking potentially missing subtle incorrect design behaviours or security vulnerabilities that are not obvious in the design source code. In fact, there is a lack of a systematic study comparing the effectiveness of different commercial and open-source LLMs in generating valid assertions from hardware design source code. AssertionBench aims to fill in the gap and provides novel insights for future research on LLMs for assertion generation.



(a) (b)

Fig. 9: **Comparison of accuracy of generated assertions**. (a) Assertion accuracy comparison for finetuned CodeLLaMa 2. (b) Assertion accuracy comparison for finetuned LLaMa3-70B. **CEX**: Counter Example trace.

IX. Limitations

- **Dataset**: In this study, we focus on Verilog designs, given its predominance in hardware design language. Moving forward, it will be intriguing to develop benchmarks for assertions in other HDLs, *e.g.*, VHDL, SystemC, to expand the scope of our analysis to broader design paradigms.
- **Modeling**: In this paper, we assessed the assertion generation capabilities of k-shot and finetuned SOTA LLMs. There is a considerable scope for improvement in terms of assertion quality and correctness. Future work should focus on modeling to capture design coverage of generated assertions and quantify their goodness in terms of captured design behavior.
- **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.

X. Conclusion and Future Work

This work introduces AssertionBench to evaluate the current and future commercial and open-source LLMs for the assertion generation task and AssertionLLM to fully automate assertion generation using generative AI without the designer's iterative intervention. Although there is no LLM that consistently outperforms other LLMs, we notice several promising trends and research directions such as (i) to quantify the goodness of assertion in terms of captured design behavior, (ii) to quantify the design coverage of the assertions, (iii) to model and capture likely design security vulnerabilities as assertions, and (iv) going beyond temporal/sequential assertions to generate assertions encompassing richer set of SVA, to enhance the practical applicability of LLMs for assertion generation task for industrial-scale designs. Pursuing these directions will further accelerate SoC and hardware design verification.

References

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

[2] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful Techniques for The Automatic Generation of Invariants. *Int'l Conf. on Computer-Aided Verification (CAV)*, 1996.

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

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

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

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

[7] A. Hekmatpour and A. Salehi. Block-based Schema-driven Assertion Generation for Functional Verification. *Asian Test Symp. (ATS)*, 2005.

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

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

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

[11] GoldMine. GoldMine: An Automatic Assertion Generation Tool. http://goldmine.csl.illinois.edu/, 2024. Accessed: January 24, 2025.

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

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

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

[15] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. AssertionBench: A Benchmark to Evaluate Large-Language Models for Assertion Generation for Hardware Design. *Conf. of the Nations of the Americas Chapter of the Assoc. for Computational Linguistics (NAACL Findings)*, 2025.

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

[17] SystemVerilog. 1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. https://ieeexplore.ieee.org/document/8299595, 2024. Accessed: January 24, 2025.

[18] Cadence. JasperGold Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2024. Accessed: January 24, 2025.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *Int'l Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Conf. on Empirical Methods in Natural Language Processing (EMNLP Findings)*, 2020.

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

[22] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. *Int'l Conf. on Software Engineering (ICSE)*, 2021.

[23] OpenCores. https://opencores.org/, 2024. Accessed: January 24, 2025.

[24] CLOC. https://github.com/AlDanial/cloc. Accessed: January 24, 2025.

[25] NCSA. NCSA Delta. https://www.ncsa.illinois.edu/research/project-highlights/delta/, 2024. Accessed: January 24, 2025.

[26] HuggingFace. Model Repository. https://huggingface.co/, 2024. Accessed: January 24, 2025.

[27] 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. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. *arXiv*, 2023.

[28] 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. Code Llama: Open Foundation Models for Code. 2024.

[29] Meta. Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/, 2024. Accessed: January 24, 2025.

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

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

[32] Samuele Germiniani and Graziano Pravadelli. 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)*, 2022.

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

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

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

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

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *arXiv*, 2023.

[38] Google Gemini. https://gemini.google.com/app. Accessed: January 24, 2025.

[39] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2023.

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

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

[42] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs. *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024.

[43] Bhabesh Mali, Karthik Maddala, Sweeya Reddy, Vatsal Gupta, Chandan Karfa, and Ramesh Karri. ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation. *IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, 2024.

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