

PALM: Program Analysis and LLM Methods for Crafting SystemVerilog Assertions

Raheel Afsharmazayejani and Benjamin Tan

Department of Electrical and Software Engineering, Schulich School of Engineering, University of Calgary

{raheel.afsharmazayej, benjamin.tan1}@ucalgary.ca

Abstract—A promising approach for security verification of a Register-Transfer Level (RTL) design is assertion-based verification (ABV), where desired properties are expressed as SystemVerilog Assertions (SVAs). To create assertions, verification engineers typically start with identifying the relevant modules and necessary variables that are relevant to a given property and then construct the assertion based on those variables. While there have been several attempts to automate assertion creation, prior work identified that automatically recognizing relevant modules and subsequently extracting the required variables within the found module to construct an SVA is a bottleneck. Recently, Large Language Models (LLMs) have emerged, demonstrating promising code generation capabilities. However, their application in helping to automate valid SVA generation, along with the combination of static analysis methods, remains not well explored. This work investigates whether, and to what extent, LLMs can assist in each stage of the automation pipeline or whether their promise requires more evidence to substantiate. This study identifies specific areas where Large Language Models (LLMs) yield measurable and practical improvements in a hybrid workflow, as well as areas where their limitations are evident.

Index Terms—Assertion-Based Verification (ABV), Security Properties, SystemVerilog Assertion (SVA), Formal Property Verification (FPV), Large Language Models (LLMs).

I. INTRODUCTION

Assertion-Based Verification (ABV) by writing high-quality SystemVerilog Assertions (SVAs) is chronically bottlenecked due to its inherent errors, the time and expertise required to write them. The process of crafting an SystemVerilog Assertion (SVA) is typically manual, where, starting from a plain language description of a property, a verification engineer needs to navigate a given implementation to find the relevant modules and signals for an assertion. In recent years, Large Language Models (LLMs) have emerged as a promising technology to accelerate and support many tasks in the digital design flow, including generating Register-Transfer Level (RTL) code [1], fixing bugs [2], and creating test benches [3].

Researchers have observed that there is an opportunity for LLMs to assist in niche domains in computer-aided design, especially in hardware security, where expertise is often scarce [4]. To improve security, prior work has suggested the

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2022-03027]. Cette recherche a été financée en partie par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG). This research work is partly supported by a gift from Intel Corporation. This work does not in any way constitute an Intel endorsement of a product/supplier. We acknowledge CMC Microsystems for provisioning products and services that facilitated this research.

adoption of a Security Development Lifecycle (SDL) that comprises security assessment, architecture review, design review, implementation review, and penetration testing [5]. Within this context, while LLMs have been explored for performing security-relevant tasks [2], [6]–[9], exploring all the myriad ways to leverage LLMs remains an open problem.

Prior deterministic-oriented work, such as [10], has attempted to automate parts of this process using program analysis-based techniques. Broadly speaking, such approaches involve taking a security property (say, in plain language, or an SVA for a reference design), identifying/extracting the relevant module and variables (signals) from the target design, and constructing the SVA. However, a possible challenge for such approaches is that they rely on anticipating implementation patterns (e.g., from prior experience), such as knowing typical signal names for identifying relevant modules, especially to set things like similarity thresholds. These can manifest as bottlenecks in the generation process.

Regarding LLM usage, how we prepare LLM (e.g., using fine-tuning [11], RAG [12]), and how we check its response (e.g., formal coverage [13], vacuity checks [14]), changes outcomes dramatically. We can use a hybrid approach as a means, not as a goal, which is currently looked at in some recent work in the literature. Therefore, the right question is not LLM vs. traditional, but which hybrid strategy yields the best result and unveils each method’s drawbacks at each stage of the framework. Thus, the primary purpose of this work is to investigate further and argue against a one-size-fits-all LLM-everywhere recipe. Overall, our contributions are as follows.

- We provide insights showing the need for rigorous, in-depth investigations before adopting LLMs as a plausible technique for SVA generation in hardware security.
- We propose a hybrid static analysis + LLM pipeline that automatically generates security-oriented SystemVerilog Assertions.
- We systematically vary where and how LLMs are used across the pipeline to identify the stages where they deliver the most value.
- We provide an open-source implementation of our approach here: [<https://github.com/CalgaryISH/PALM>].

II. BACKGROUND

Prior work in assertion generation methods can be considered along different perspectives, including the stage in the design life-cycle when SVAs are generated (RTL or Pre-RTL), method

TABLE I: Comparative analysis between PALM and existing SVA generation methods.

Method	Stage	Target	Approach	SVA Gen.	Agentic	LLM Stage Efficacy	Evaluation Resources
Kande et al. [9]	RTL	Security	LLM-based	✓	✗	✗	OpenTitan; Hack@DAC'24; Manually crafted
AssertLLM [15]	Pre-RTL	Functionality	LLM-based	✓	✗	✗	I2C; ECG; Pairing
Spec2Assertion [16]	Pre-RTL	Functionality	LLM-based	✓	✗	✗	UART; SOCKET; I2C; HTAX
FLAG [13]	Pre-RTL	Functionality	Hybrid	✓	✗	✗	AXI; Wishbone; PCI; ABP; Q-Channel; P-Channel
Transys [10]	RTL	Security	Algorithmic	✓	✗	✗	AES; ORI200; RSA
LASP [6]	RTL	Security	Hybrid	✓	✗	✗	AES; RSA; DES; SHA; Ibex
LASA [12]	RTL	Security	LLM-based	✓	✗	✗	CEP; OpenTitan; Hack@DAC'24
This work	RTL	Security	Hybrid	✓	✓	✓	AES; RSA; SHA; FSM Controller (GitHub; TrustHub; OpenTitan; Manually crafted)

of generation (algorithm-based [10], [17]–[19], LLM-based [9], [15], [20], Natural Language Processing (NLP)-based [21], [22], or hybrid [6], [23]), and target (security or functionality). Functionality-oriented SVAs (e.g., the works done in [15], [23]–[25]) validate the correct operational behavior of the design according to the functional requirements within a design specification. These assertions ensure that the intended design operations perform correctly and that the design reacts properly to different inputs. Security-oriented SVAs (e.g., the work done in [17], [9], [6], [8], [19], [26]) address confirming and implementing security properties of a design, such as guaranteeing data confidentiality, integrity, and availability, or focusing on the functionality of security-related designs. Recent work [27] on generating assertions for the open-source System-on-Chips (SoCs) observed that while plenty of designs exist, few security properties are publicly available.

Some recent works explore different ways to combine LLMs with analysis signals. Spec2Assertion [16] proposes a method at the pre-RTL stage and aims at functional assertions. It creates a Signal Dependency Graph (SDG) over a golden RTL to calculate an assertion importance score. They use this graph analysis only after assertion generation, for ranking and evaluation purposes. In LASP [6], the authors do not provide sufficient technical detail to enable us to pinpoint the effectiveness of static analysis at each stage. Table I depicts a comparative analysis between PALM (this work) and existing SVA generation methods in the literature.

Although all SVA generation methods that use LLM try to enhance their assertions by different strategies from using static analysis as a tool to help the evaluation process to using RAG and fine-tuning strategies - thus far, the trend in the literature has been to focus on improving LLMs for generating SVAs without considering whether LLMs are best suited for sub tasks within the flow. Thus, our work investigates LLM suitability for steps within a generation process instead of simply end to end.

III. METHOD

A. Motivation: from plain prompting to an agentic pipeline

We initially investigated a direct LLM workflow that chained two prompt generators: (1) an Analyzer prompt that asked the model to extract the target module and relevant signals from the RTL design code, and (2) a Generator prompt that asked the model to construct the SVA and TCL files from the caught signals. Despite careful prompt engineering, this setting surfaced

two significant issues: non-code text and comments appearing in outputs, and the need for human fixes at each step to remove extra text and make the files ready for the evaluation step. These drawbacks, in terms of both quality and cost/latency, motivated us to follow a shift to an agentic framework that modularizes responsibilities into several expert agents (each with a specific responsibility), comprises LLM-based strategies with program analysis, and automates refinement/iteration while constraining outputs to a given skeleton. Moreover, this strategy enables us to identify LLM and program analysis in various roles, which, as a necessary approach, facilitates finding the appropriate role match.

B. Overall Framework

We propose a multistage pipeline (shown in Figure 1) that combines LLM-driven and program-analysis-based techniques to automatically generate and evaluate SystemVerilog Assertions (SVAs) for different IP types. Beyond automation, the framework we designed is an instrument to let us study when and where LLMs help in the process of automating assertion generation. For this purpose, we pose the following research questions:

RQ1: At what stage of the SVA generation pipeline do LLMs have a valuable impact compared to their program analysis counterpart?

RQ2: What is the cost and quality trade-off of different PA/LLM placements?

The proposed framework stages include:

- 1 Provision:** In the first stage, a security property written in natural language is linked to the specific module or function of the design it references. Then, we craft a corresponding template for that particular property. After that, we feed different sets of multi-module designs and a library of security properties. We run several program analysis steps to enhance the design with a machine-readable and better-to-analyze structure. These analyses include: (1) Parsing HDL design code and converting it into Abstract Syntax Tree (AST) format. (2) AST extraction (module names, ports and variable types and directions, signal bit width, and so on). (3) Program Dependence Graph (PDG) [28] construction, which uses the control and data dependencies of the variables. (4) Using PDG summaries to rank signals or variables by their influence within the design (e.g., depth and centroid). These artifacts

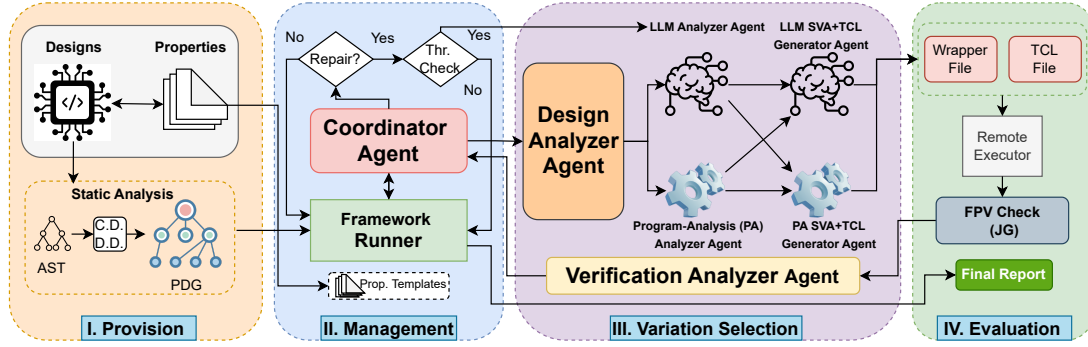


Fig. 1: An Overview of PALM SVA Generation Pipeline.

seed later agents in the following stage with deterministic evidence (module lists and candidate signals), reducing guesswork by the LLM (if the LLM gets involved in the pipeline) or facilitating the extraction of target variables and constructing the SVA using a program-analysis-based method (if this strategy is used).

- 2 **Management:** In this stage, the Coordinator Agent orchestrates the pipeline end-to-end. The primary responsibility of the coordinator is to direct the pipeline to the heart of the framework in the following stage: the Design Analyzer agent, which selects analyzing and generating approaches. This agent also handles the refinement loop to mitigate the errors arising from the formal tool evaluation process. The Verification Analyzer Agent analyzes the results of a formal tool to determine if the property is proven or not. Furthermore, in an LLM strategy, if the log file includes errors, it provides another opportunity to repair it, facilitating its retransmission to the refinement loop to address the existing issue within the generated log file.
- 3 **Variation Selection:** The current stage operates based on the selected strategy to analyze the code (by using the generated artifacts from the first stage) and then construct SVAs. It is possible to choose an analysis strategy (LLM-based or Program-Analysis-based), which is then followed by a generation strategy (LLM-based or Program-Analysis-based), based on the design category and its corresponding properties.
- 4 **Evaluation:** In the final stage, the framework feeds the generated Wrapper and TCL into a remote formal property verification tool through the FPV check component. The generated log result will get back to the Verification Analyzer Agent. If the property evaluation shows the proven (pass) or non-proven (fail), the final report reflects them. If an error exists in the log, when employing an LLM-based methodology, it is possible to feed the log file back to the LLM and request refinement. For this purpose, we set a threshold of five attempts to enable improvement of the quality of assertions and the TCL file.

C. Design Variations

Variation 1: Program-analysis-based method for both design analysis and assertion generation (PA-PA). In this variant, every stage of the pipeline is driven by program analysis artifacts, ensuring a completely deterministic flow.

Through this step, provision converts the HDL main design (MDesign) into an Abstract Syntax Tree (AST) format by using the Slang tool [29].

Then, the algorithm iterates over all modules in MDesign and compares them to a predefined set of possible function names. The comparison uses the Sorensen Dice Index (SDI) [30], a similarity metric that quantifies how closely a given module name string matches an expected function name. Then, the algorithm selects the module with the best module candidate that is relevant to the target functionality. This approach filters out unrelated module names; however, if the property or function is implemented in another structure (e.g., a function or a code snippet) instead of a "module" structure, it is a challenge for the algorithm to find the target module.

This recognition process of the key variables from the extracted module involves analyzing variables in the main HDL design (MDesign) and an assistant HDL design (ADesign). Another similar HDL (ADesign) design is used to assist the main design in this step. We have used ADesign to provide a minimal function equivalent to the property, helping to prune out irrelevant variables by using a cross-comparing strategy. The extraction strategy uses statistical, structural, and semantic analysis, inspired by the Transys [10] approach, to compute a weighted similarity score for variable pairs of extracted modules across MDesign and ADesign. This score is used as a threshold-based pruning mechanism to help ensure that only variables with high statistical, structural, and semantic relevance remain. However, the threshold score mechanism cannot filter out all key variables; therefore, we used one or more heuristic steps to find the required variables for the templates. These steps include verifying variable size and type consistency, conducting signal role checks using a constructed subPDG, and, when necessary, identifying clock and reset signals via lightweight regex-based name matching.

Provision ultimately builds per-module PDGs and exports two products: (1) a summary list that maps each design to the most likely module per family/function. (2) one or more property lists that list the variables/signals with various features like direction/width, PDG depth, centroid, and operator counts. By using these artifacts, the algorithm chooses the DUT and maps property roles to real signals necessary for the final SVA. In the following step, the algorithm replaces a family-specific template's variables with the chosen module and signals to cre-

ate a wrapper (instantiates DUT, connects clk/reset, references everything else hierarchically) and a TCL (analyze, elaborate, set clock/reset, prove -all).

In the next step, using a remote execution tool, the design and all generated artifacts are sent and evaluated by an FPV tool (Cadence JasperGold). Then, the log report will be sent back to the verification analyzer agent. And this agent sends the final report to the coordinator agent.

Variation 2: LLM-based method for both design analysis and assertion generation (LLM-LLM). In this variant, all major decisions are made using language models. During Provision, we present the HDL main design (MDesign) to an analysis agent that is specialized per IP family/property. The LLM reads the RTL and returns a roles table and a DUT guess (top module to bind the property).

To limit extra generating text and verbosity, the agent forces a single fenced block containing only a short description paragraph, a Markdown roles table with fixed headers (rolecandidate(s) in designchosenwidthlnotes). A JSON/CSV extractor then normalizes the format of this block. If required, cells are missing or malformed, the agent requests a self-repair pass from the LLM (e.g., filling empty cells, maintaining the structure, and omitting extraneous prose).

SVA + TCL Generation agent uses the roles and the design snippet to compose the wrapper, SVA, and TCL. This agent is template-aware and uses a template-based prompting strategy to avoid inventing new parts for each generated assertion. The agent then required to instantiate the DUT as `u_dut`, connect `clk/reset` at the wrapper ports, reference all other signals hierarchically (`u_dut.sig`), avoid comments and extraneous prose, and output exactly three blocks in order: wrapper, SVA block, TCL script. A post-processor enforces these structure rules and removes any extra commentary. Similar to the first variation, a remote executor operates the FPV check. If the JasperGold log report contains errors (e.g., missing module, syntax issues, and so on), a repair agent prompts the LLM with the failure section and the prior artifacts to regenerate the faulty part, keeping the rest unchanged. The agent sets a threshold to control the number of repair attempts.

Variation 3: Program-analysis-based approach for design analysis and LLM-based method for assertion generation (PA-LLM). This hybrid variation keeps the front half deterministic while letting a generator LLM do the rest of the process. In Provision, an algorithm parses the HDL and makes AST and PDG artifacts. The program-analysis analyzer agent then determines the DUT and candidate signals that play key roles in achieving the target property. Then the framework requests the LLM to construct and return the wrapper, SVA, and the TCL file. The subsequent steps are similar to the second variant, which is equipped with a self-refinement log procedure.

Variation 4: LLM-based method for design analysis and Program-analysis-based method for assertion generation (LLM-PA). In contrast to the third variation, in the Provision stage, the HDL is available to the LLM analysis agent, which returns a role table (DUT guess, candidate signals, notes) in a fenced format. The algorithm scans the generated response to

TABLE II: Properties and related CWEs per design family.

Designs	#	Property	Description	Relevant CWE
AES	11	AES_P1	The subkey is applied by XOR between each state byte and the matching subkey byte.	CWE-325
		AES_P2	The round constant must be correctly applied in each key-expansion round.	CWE-325
		AES_P3	The S-box must avoid fixed points and opposite fixed points.	CWE-1240
		AES_P4	ShiftRows cyclically shifts each row by its specified offset.	CWE-325; CWE-682
RSA	3	RSA_P1	Output cipher must differ from input message when <code>start</code> rises.	CWE-1431; CWE-1240
SHA	3	SHA_P1	Consecutive differing accepted inputs must produce differing outputs.	CWE-328; CWE-1240
FSM	5	FSM_P1	After reset, every clock edge: state must be one of the declared legal encodings (no <code>X/Z</code>).	CWE-1245
		FSM_P2	If an illegal or <code>X/Z</code> state occurs after reset, next state must go to a SAFE state (e.g., <code>IDLE</code>).	CWE-1245

Syntax Correctness and FPV Check Per Each Variation

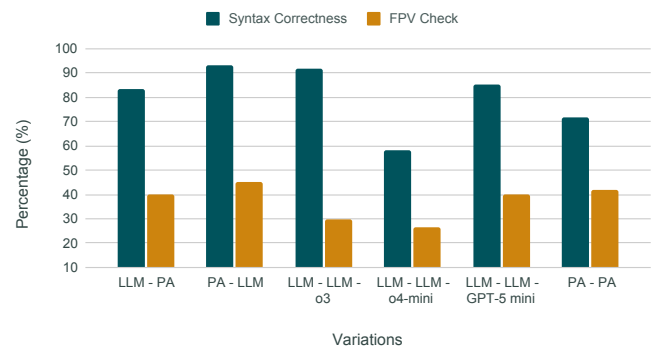


Fig. 2: The overall average percentage of syntax correctness and FPV pass for all designs and all properties per each Variation. For LLM-PA and PA-LLM variants, we used the default LLM (o4-mini).

extract the variables for use in the SVA template replacement. The evaluation stage is similar to the first variants.

IV. EXPERIMENTAL EVALUATION

A. Benchmarks and Setup

To investigate the efficiency of our proposed framework, we have implemented a Python-based framework that includes all four variation modes on a backbone of an agentic infrastructure: PA-PA, LLM-PA, PA-LLM, LLM-LLM. For LLM-LLM variants, we compare three models (by OpenAI [31]): `gpt-4o-mini`, `gpt-5-mini`, and one reasoning model (`o3`) while other variations use `gpt-4o-mini` as the default LLM.

For comparison, we implemented, as a baseline, a Python-based non-LLM-based approach that uses program analysis techniques built with a SystemVerilog front-end. As benchmarks, we curated a set of 22 open-source implementations in SystemVerilog, including 11 AES crypto cores, three RSA designs, three SHA, and five FSM controllers. The set of benchmark RTL designs is collected from public GitHub repositories, TrustHub [32], some OpenTitan SoC [33] controller modules, and some designs manually crafted. The property suite is summarized in Table II. Properties used for AES cores are the same security essential checks used in Transys [10]. For each, we identified the most relevant weaknesses from hardware common weakness enumeration (CWE) [34] list (e.g., CWE-325, CWE-1240, CWE-682). The property of RSA aligned again with the property used in [10], could be assigned to CWE-1431

TABLE III: LLM Efficacy Score (LES) Calculation by Variant

Variant	Syntax correctness		FPV Check Pass		Token		Time		Repair Attempts		LES
	Syntax (%)	Normalized	FPV (%)	Normalized	Token (#)	Normalized	Time (sec)	Normalized	Count	Normalized	
PA-PA	71.66	0.38	41.66	0.81	0	1	43	1	0	1	0.75
PA-LLM	93.33	1	45	1	1916207	0.64	1690	0.48	15	0.94	0.87
LLM-PA	83.33	0.71	40	0.72	162754	0.96	349.8	0.90	0	1	0.81
LLM-LLM	58.33	0	26.66	0	5413424	0	3254.21	0	255	0	0

or CWE-1240. SHAP1 is the corresponding property check for CWE-328 or CWE-1240. And finally, both properties for FSM controllers are relevant to detecting potential weaknesses regarding CWE-1245.

To characterize the performance of our proposed LLM-based SVA pipeline, we evaluated the syntactical correctness of the generated SVAs (Syntax) as well as whether the generated SVAs correctly represent the expected behavior of the design (FPV). Given that our benchmark implementations are all correct, if an assertion fails FPV, it indicates that there is a misalignment between the generated SVA and the expected hardware behavior.

Figure 2 shows six variation (LLM-PA, PA-LLM, LLM-LLM-o3, LLM-LLM-GPT-4o-mini, LLM-LLM-GPT-5-mini, and PA-PA) for total 60 tests per setting (AES: 11 designs \times 4 properties = 44; RSA: 3 \times 1 = 3; SHA: 3 \times 1 = 3; FSM: 5 \times 2 = 10). Regarding this Figure, the green bar represents the percentage of syntax-correct assertion generations, and the orange bar depicts the percentage of FPV passes over the total number of tests (60). Across all settings, PA-LLM produces the best overall outcome, with 93% for syntax correctness and approximately 45% for FPV pass. Front-loading program analysis performs effectively to pick the DUT and role signals, resulting in removing most naming and ambiguity. At the same time, wrapper/SVA/TCL construction by LLM can provide the flexibility to compose headers and hierarchical references correctly. The LLM-LLM-o3 got the second rank for Syntax (around 91%), with a relatively low FPV percentage. The lowest FPV check belongs to LLM-LLM-o4-mini (around 26%). The fully deterministic PA-PA baseline achieves approximately 71% syntax and 41% FPV.

Figure 3 shows the average percentage of syntax correctness and FPV check percentage for all given IP families and all given properties (AES_P1-AES_P4, RSA_P1, SHA_P1, FSM_P1-FSM_P2). Figure 3(a) shows that, for every property except AES_P1, the generated SVAs are syntactically correct. RSA_P1, and both properties of the FSM IP family could reach 100% of syntax correctness through all pipeline variations. Although the PA-PA strategy is one stable and deterministic method, it surprisingly shows competitive results compared to other variants. For example, AES_P2 provides better results for FPV than LLM-PA and all other pure LLM framework variations. Based on the LLM-LLM methodology for the o3 model, we achieved the second-highest level of syntax correctness; however, this reasoning model was unable to prove that it could generate SVAs that successfully passed the FPV check step. Comparing this model (o3) results to other LLM models (gpt-4o-mini and gpt-5-mini), the results demonstrate that only some cases have better performance in terms of both

syntax correctness and FPV check. Among all models, we could capture the best average results from gpt-5-mini model in terms of FPV check.

In addition to the metrics mentioned, and to address the primary purpose of this article, which is to investigate the efficacy of the LLM stage, we proposed a metric that can be used in hybrid methods and different variations in terms of using or not using LLM to assess the influence of LLM. Toward this end, we evaluated each variation using a composite LLM Efficacy Score (LES). For constructing this score, we aggregate FPV success, tokens, generation and evaluation time, and repair iterations (attempts to repair the erroneous logs through the following formula. LES uses a weighted sum, using multi-attribute construction, then following normalization across variants, and finally calculating the formula for each variation. The formula we suggest is as follows:

$$LES_v = \alpha F_v + \beta S_v + \gamma C_v + \delta R_v + \varepsilon T_v, \quad (1)$$

$$\alpha, \beta, \gamma, \delta, \varepsilon \geq 0, \quad \alpha + \beta + \gamma + \delta + \varepsilon = 1,$$

$$F_v = \text{Norm}^+(\text{FPV}_v), \quad S_v = \text{Norm}^+(\text{Syntax}_v),$$

$$C_v = \text{Norm}^-(\text{Token}_v), \quad R_v = \text{Norm}^-(\text{Repairs}_v),$$

$$T_v = \text{Norm}^-(\text{Time}_v).$$

F_v (or $\text{Norm}^+(\text{FPV}_v)$) and S_v (or $\text{Norm}^+(\text{Syntax}_v)$) normalize FPV pass rate and syntax correctness, respectively (the higher is the better). Moreover, C_v (or $\text{Norm}^-(\text{Token}_v)$) normalized token consumption (lower is better). R_v (or $\text{Norm}^-(\text{Repairs}_v)$) normalized repair attempts (lower is better). And finally T_v (or $\text{Norm}^-(\text{Time}_v)$) normalized total time of SVA generation (lower is better). When assigning values to parameters ($\alpha, \beta, \gamma, \delta$, and ε), we selected the most significant value for α and β since we chose to give the heavier weights (0.3) to the FPV pass and Syntax correctness. Then, for γ , we chose 0.2 as the second prominent feature. And for the coefficients for time and repair weights, we selected 0.1. Based on the attained statistics from four variations, we computed LES for each SVA generation variations. The results are shown in Table III. As can be seen from the table, the LLM efficacy score of PA-LLM variants outperforms the other variations. As the second-best performer, the LLM-PA variant achieves the LES of 0.81.

V. DISCUSSION

1- Regarding our findings in this work, it is too early to solely rely on LLMs as a standalone solution in the digital design flow. However, results from PA-LLM showed that the self-refinement strategy, applied after formal verification result analysis, can enhance the final results. Therefore, there is a need to shift the focus from improving LLM results to embedding

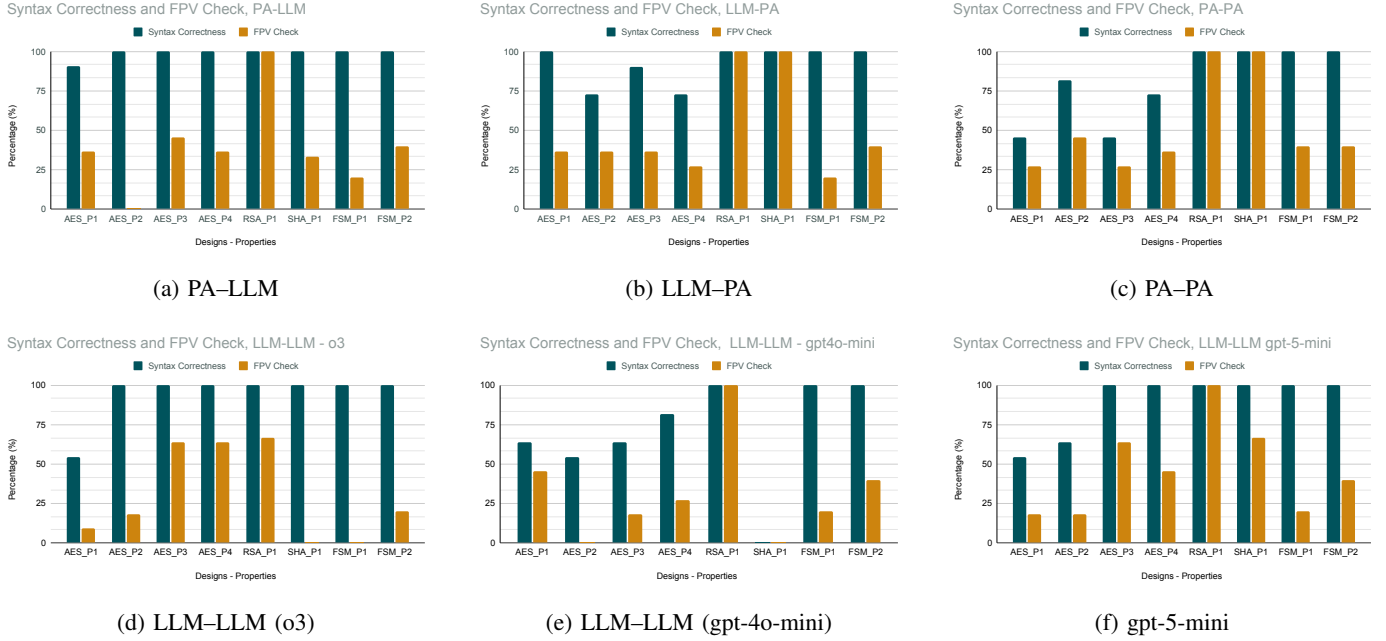


Fig. 3: Syntax Correctness and FPV Check Percentage Per Design, Per Property.

LLM in traditional methods, thereby maintaining the trade-off between performance and cost.

2- As an insight, we think LES can be used as a standard assessment factor to evaluate LLM placement in other hardware flows (e.g., bug detection, RTL code generation, and so on). Forthcoming versions of LES can comprise more task-specific weights.

3- We manually investigated some of the log, SVA, and TCL-generated files. We observed the following takeaways, which may be helpful for other researchers and can be used for the augmented version of the current agentic framework:

- In the PA-LLM variations, we found some errors like "reset' is not declared" and "clk' is not declared" in the first generated log file by the FPV tool. This error shows that the LLM was not able to add these signals to the first wrapper or TCL file. But in its subsequent refinement, it could recognize this issue and revise the wrapper or TCL code accordingly. Thus, a self-refinement process can play a significant role in detecting some problems in the first construction attempts.
- For two properties, we ended up with zero performance in terms of syntax correctness and FPV check, under the pure LLM strategy (with two different LLMs). The reason for it was that a single missing parenthesis cascaded into the subsequent generated wrappers. And the same error perpetuates in every refinement loop, and even the LLM repair agent was not able to catch it. This is exactly the kind of case that a small prompting upgrade could fix, e.g., adding an explicit bracket/parenthesis-balance check to the constraint list before regenerating.
- Compared to our motivation case with using two sequences of prompt generators and without using the agen-

tic strategy, we found that the proposed agentic approach exhibited less hallucination compared to the motivating example. This could be due to our engineering strategy to request that the LLM shape its outputs with a predefined template, thus constraining the output. One other possible option that can be investigated in future work is using such a constraint at the final stages of the pipeline to check some post-annotation checks, like comparing the final signal list with the generated ones with the PA method, to ensure the correct result and avoid extra cost by entering the refinement loop.

- For cases where the FPV fails, one potential solution for the next version of this framework could be using the visual format (waveform) of the exact instances and the signal that violates the property. Investigate the impact of feeding this piece of information to LLM, as one other feature for the refinement process could be the next future purpose.

VI. CONCLUSION

In this work, we proposed an agentic framework that combines program analysis (PA) with large language models (LLMs) to generate SystemVerilog Assertions. Furthermore, we provided investigations and insight to when and how LLMs contribute into that automation process. We also introduced the LLM Efficacy Score (LES). This compact, quantitative measurement integrates the effectiveness of Syntax correctness and FPV passes, efficiency of used input and output tokens fed to or received from LLM, total time of completing the tasks, and the impact of repair iterations into a single, comparable number per variant. In our experiments, we found that LES of PA-LLM variation offers the best quality-per-cost trade-off among all other pipeline variations.

REFERENCES

- [1] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, apr 2023, pp. 1–6, ISSN: 1558-1101.
- [2] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On Hardware Security Bug Code Fixes by Prompting Large Language Models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10462177/>
- [3] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, ser. MLCAD '24. New York, NY, USA: Association for Computing Machinery, sep 2024, pp. 1–10. [Online]. Available: <https://dl.acm.org/doi/10.1145/3670474.3685956>
- [4] H. Pearce and B. Tan, "Large Language Models for Hardware Security (Invited, Short Paper)," in *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, oct 2024, pp. 420–423. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10835653>
- [5] H. Khattri, N. K. V. Mangipudi, and S. Mandujano, "HSDL: A Security Development Lifecycle for hardware technologies," in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. San Francisco, CA, USA: IEEE, Jun. 2012, pp. 116–121. [Online]. Available: <http://ieeexplore.ieee.org/document/6224330/>
- [6] A. Ayalasomayajula, R. Guo, J. Zhou, S. K. Saha, and F. Farahmandi, "LASP: LLM Assisted Security Property Generation for SoC Verification," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. Salt Lake City UT USA: ACM, sep 2024, pp. 1–7. [Online]. Available: <https://dl.acm.org/doi/10.1145/3670474.3685967>
- [7] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "LLM for SoC Security: A Paradigm Shift," *IEEE Access*, vol. 12, pp. 155 498–155 521, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10596266/>
- [8] S. Paria, A. Dasgupta, and S. Bhunia, "DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection," aug 2023, arXiv:2308.06932 [cs]. [Online]. Available: <http://arxiv.org/abs/2308.06932>
- [9] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "(Security) Assertions by Large Language Models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4374–4389, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10458667/>
- [10] R. Zhang and C. Sturton, "Transys: Leveraging Common Security Properties Across Hardware Designs," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, may 2020, pp. 1713–1727. [Online]. Available: <https://ieeexplore.ieee.org/document/9152775/>
- [11] M. Shahidzadeh, B. Ghavami, S. J. E. Wilton, and L. Shannon, "Automated Verilog Assertion Generation Using Fine-Tuned LLMs with Subtask-Specific Iterative Prompting," in *2025 26th International Symposium on Quality Electronic Design (ISQED)*. San Francisco, CA, USA: IEEE, Apr. 2025, pp. 1–7. [Online]. Available: <https://ieeexplore.ieee.org/document/11014349/>
- [12] D. R. Ankireddy, S. Paria, A. Dasgupta, S. Ray, and S. Bhunia, "LASA: Enhancing SoC Security Verification with LLM-Aided Property Generation," Jun. 2025, arXiv:2506.17865 [cs]. [Online]. Available: <http://arxiv.org/abs/2506.17865>
- [13] Y.-A. Shih, A. Lin, A. Gupta, and S. Malik, "FLAG: Formal and LLM-assisted SVA Generation for Formal Specifications of On-Chip Communication Protocols," Apr. 2025, arXiv:2504.17226 [cs]. [Online]. Available: <http://arxiv.org/abs/2504.17226>
- [14] L. Di Guglielmo, F. Fummi, and G. Pravadelli, "Vacuity analysis for property qualification by mutation of checkers," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. Dresden: IEEE, Mar. 2010, pp. 478–483. [Online]. Available: <http://ieeexplore.ieee.org/document/5457158/>
- [15] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, Z. Xie, and H. Zhang, "AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs," nov 2024, arXiv:2402.00386 [cs]. [Online]. Available: <http://arxiv.org/abs/2402.00386>
- [16] F. Wu, E. Pan, R. Kande, M. Quinn, A. Tyagi, D. K. Houngrinou, J. Rajendran, and J. Hu, "Spec2Assertion: Automatic Pre-RTL Assertion Generation using Large Language Models with Progressive Regularization," May 2025, arXiv:2505.07995 [cs]. [Online]. Available: <http://arxiv.org/abs/2505.07995>
- [17] N. F. Dipu, A. Ayalasomayajula, M. M. Tehranipoor, and F. Farahmandi, "AGILE: Automated Assertion Generation to Detect Information Leakage Vulnerabilities," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 1794–1809, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10363345/>
- [18] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "AutoSVA: Democratizing Formal Verification of RTL Module Interactions," apr 2021, arXiv:2104.04003 [cs]. [Online]. Available: <http://arxiv.org/abs/2104.04003>
- [19] C. Wang, Y. Cai, Q. Zhou, and H. Wang, "ASAX: Automatic security assertion extraction for detecting Hardware Trojans," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. Jeju: IEEE, jan 2018, pp. 84–89. [Online]. Available: <http://ieeexplore.ieee.org/document/8297287/>
- [20] A. Menon, S. S. Miftah, A. Srivastava, S. Kundu, S. Kundu, A. Raha, S. Banerjee, D. Mathaikutty, and K. Basu, "OpenAssert: Towards Secure Assertion Generation using Large Language Models," in *2025 IEEE 43rd VLSI Test Symposium (VTS)*. Tempe, AZ, USA: IEEE, Apr. 2025, pp. 1–5. [Online]. Available: <https://ieeexplore.ieee.org/document/11022798/>
- [21] J. Zhao and I. G. Harris, "Automatic Assertion Generation from Natural Language Specifications Using Subtree Analysis," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Florence, Italy: IEEE, mar 2019, pp. 598–601. [Online]. Available: <https://ieeexplore.ieee.org/document/8714857/>
- [22] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, "ChIRAAAG: ChatGPT Informed Rapid and Automated Assertion Generation," jun 2024, arXiv:2402.00093 [cs]. [Online]. Available: <http://arxiv.org/abs/2402.00093>
- [23] F. Aditi and M. S. Hsiao, "Hybrid Rule-based and Machine Learning System for Assertion Generation from Natural Language Specifications," in *2022 IEEE 31st Asian Test Symposium (ATS)*. Taichung City, Taiwan: IEEE, nov 2022, pp. 126–131. [Online]. Available: <https://ieeexplore.ieee.org/document/9979024/>
- [24] R. Krishnamurthy and M. S. Hsiao, "EASE: Enabling Hardware Assertion Synthesis from English," in *Rules and Reasoning*, P. Fodor, M. Montali, D. Calvanese, and D. Roman, Eds. Cham: Springer International Publishing, 2019, vol. 11784, pp. 82–96, series Title: Lecture Notes in Computer Science.
- [25] C. B. Harris and I. G. Harris, "GLAsT: Learning Formal Grammars to Translate Natural Language Specifications into Hardware Assertions," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016, pp. 966–971. [Online]. Available: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7459447>
- [26] H. Witharana, A. Jayasena, A. Whigham, and P. Mishra, "Automated Generation of Security Assertions for RTL Models," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 19, no. 1, pp. 1–27, jan 2023. [Online]. Available: <https://dl.acm.org/doi/10.1145/3565801>
- [27] J. Rogers, N. Shakeel, D. Mankani, S. Espinosa, C. Chabra, K. Ryan, and C. Sturton, "Security Properties for Open-Source Hardware Designs," dec 2024, arXiv:2412.08769 [cs]. [Online]. Available: <http://arxiv.org/abs/2412.08769>
- [28] J. Ferrante, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, 1987. [Online]. Available: <https://sv-lang.com/>
- [29] "slang C++ docs." [Online]. Available: <https://sv-lang.com/>
- [30] L. R. Dice, "Measures of the Amount of Ecologic Association Between Species," *Ecology*, vol. 26, no. 3, pp. 297–302, jul 1945. [Online]. Available: <https://esajournals.onlinelibrary.wiley.com/doi/10.2307/1932409>
- [31] OpenAI, "ChatGPT: Optimizing Language Models for Dialogue," nov 2022. [Online]. Available: <https://openai.com/blog/chatgpt/>
- [32] "Trust-Hub.org." [Online]. Available: <https://trust-hub.org/#/home>
- [33] "Open source silicon root of trust (RoT) | OpenTitan." [Online]. Available: <https://opentitan.org/>
- [34] T. M. C. (MITRE), "CWE - CWE-Compatible Products and Services," Dec. 2020. [Online]. Available: <https://cwe.mitre.org/compatible/compatible.html>