Modular End-to-End Pipeline for Formal Property Verification using Large Language Models

Luca Müller[1], Chandan Kumar Jha[2], Benjamin Arlt[2], Muhammad Hassan[1,2], Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI, Bremen, Germany
[2]Institute of Computer Science, University of Bremen, Bremen, Germany
{luca.mueller}@dfki.de {chajha, arltb, hassan, drechsler}@uni-bremen.de

***Abstract*—Formal Property Verification (FPV)** **makes the verification of industry-scale hardware designs feasible by putting the focus on desired properties that need to be satisfied. However, writing *System-Verilog Assertions (SVAs)* based on textual specifications to model these properties is time-intensive and prone to errors. This can be attributed to the verbosity and ambiguity of natural language used to specify system behavior. *Large Language Models (LLMs)* have been proposed to mitigate errors, cut costs, and reduce time-to-market during property generation. While prior works have shown the potential of LLMs in this regard, there are several limitations that need to be investigated, e.g., the consideration of only small design sizes, a lack of incorporation of formal tools, or a focus on only specific types of properties.**

In this work, we mitigate these limitations and introduce a modular, end-to-end pipeline for FPV using LLMs. The evaluation stage is directly integrated with a formal verification tool, making use of tool feedback in the loop to improve the quality of generated assertions. Our pipeline is built to act as a baseline for future research, allowing easy integration of further processing steps and refinements. We demonstrate the effectiveness of our proposed approach on two OpenTitan IP modules, evaluating both syntactic and semantic quality of LLM-generated assertions and comparing our results to a zero-shot baseline. Experimental results show that the assertion pass rate can be improved by more than 2x and formal coverage can be improved by 86% with our pipeline compared to the baseline.

I. Introduction

With the increasing size of hardware designs, new challenges arise throughout the *Electronic Design Automation (EDA)* flow [1]. Verification is especially affected, as formal methods gain relevance when exhaustive simulation becomes infeasible, but at the same time suffer from the challenge of scalability themselves [2]. *Formal Property Verification (FPV)* addresses this problem, moving away from proving correctness under all input vectors toward specific properties a design has to satisfy [3]. This approach enables formal verification of industry-scale hardware designs [4], but shifts the verification complexity to writing good *System-Verilog Assertions (SVAs)* to model these properties. In particular, questions about the high manual effort and error-sensitivity of translating textual specifications to formal properties can be raised due to the ambiguity of natural language [5].

Recently, the use of *Large Language Models (LLMs)* for this task has been extensively discussed, with the aim of reducing time-to-market and ideally mitigating errors. Shahidzadeh et al. [6] show high coverage of generated assertions but do not evaluate industry-scale designs, while ChIRAAG [7] does not report any coverage results. Paria et al. [8] and Kande et al. [9] focus on the generation of properties related to security verification. LISA [10] attempts to achieve coverage closure through iteration but is limited to the specific verification setup. A variety of other works investigate further optimizations to specific parts of the generation process, e.g., AssertLLM [11], SANGAM [12], SuperSAGA [13], AssertMiner [14], and the work by Quddus et al. [15]. Reviewing the state-of-the-art reveals a rapid development of frameworks for the integration of LLMs for FPV. However, existing works leave little room for evolution and only focus on specific parts of the workflow.

To address this gap, in this work, we present a modular, end-to-end pipeline for FPV using LLMs. It divides the process of end-to-end FPV into four distinct stages, pre-processing, generation, post-processing, and evaluation, finally allowing for an analysis of FPV results by a human expert. Pipeline stages are targeted at dealing with industry-scale hardware designs and integrate formal tools to improve the quality of generated assertions. While we propose concrete processing steps within these four stages, our modular pipeline aims to serve as a baseline for future research by allowing for easy integration of further approaches and optimizations at each stage. To evaluate the proposed workflow, we compare it with a zero-shot baseline on two industry-scale open-source OpenTitan [16] *Intellectual Property (IP)* designs. In summary, we make the following contributions.

- Introduce a modular, end-to-end FPV pipeline with prompt templates for property generation using LLMs, which can deal with industry-scale hardware designs and complex specification documents.
- Implement pipeline stages for easy extendability and directly integrate the evaluation stage into the pipeline with a formal tool in the loop.
- Demonstrate the effectiveness of our proposed pipeline by showing improvements over a zero-shot baseline on two OpenTitan IP designs, considering syntactic quality (number of valid assertions), as well as semantic quality (correctness and coverage of properties).

In the following, Section II introduces preliminary information to keep this work self-contained. Section III introduces our modular, end-to-end pipeline, which transforms a set of specification documents to a complete FPV evaluation. Section IV experimentally evaluates our proposed pipeline on two industry-scale hardware modules. Section V concludes our work and provides an outlook into future research directions.

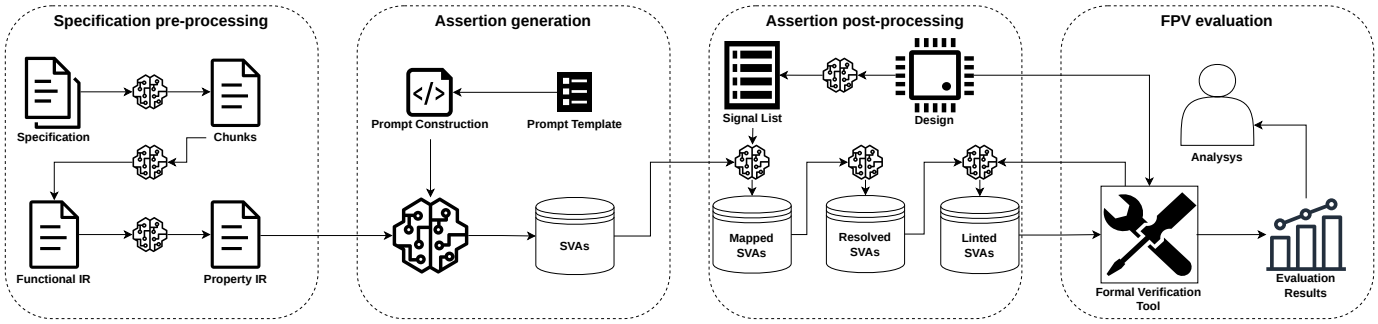

Fig. 1. Our proposed end-to-end pipeline is composed of four stages: specification pre-processing, assertion generation, assertion post-processing and FPV evaluation. It takes a set of specification documents as input and outputs a complete evaluation of FPV results to be analyzed by a human expert. Each stage is built in a modular fashion, i.e., individual processing steps can be added, removed or modified in a simple way.

II. Preliminaries

A. Large Language Models

LLMs are transformer-based *Artificial Intelligence (AI)* models designed for *Natural Language Processing (NLP)* tasks and trained on huge corpora of data. They process text in the form of tokens, utilizing the self-attention mechanism and its parameter weights to generate a coherent next output token sequence based on the previous context [17]. Input sequences are referred to as prompts, which provide a structured way to instruct the LLM to execute a given task. Prompts can be refined in several ways, e.g., by providing n example outputs via n-shot prompting, by filling run-specific data in a prompt template, or by widening the context with *Retrieval Augmented Generation (RAG)*. Some LLM providers make weights publicly available in open-weight models, enabling them to be run locally. Other providers do not disclose weights, making their closed-weight models accessible through an *Application Programming Interface (API)*. Since the rise of LLMs, various applications have been proposed in the EDA domain [18], especially in the fields of design and verification [19]–[21].

B. Formal Property Verification

FPV is a static *Assertion Based Verification (ABV)* approach that formally verifies whether all possible behaviors of a given design satisfy a given set of properties [3]. It belongs to the model checking branch of formal verification [2] and has long been established in industry use as a suitable abstraction level for formal verification [4]. The general idea of FPV is to write properties that are then asserted, i.e., they have to be satisfied by the *Register Transfer Level (RTL)* design. Properties can generally be formulated freely, but usually follow different forms, like invariants or implication properties. The language used to describe properties depends on the *Hardware Description Language (HDL)* used for the implementation of the design. Previous work on the integration of LLMs for FPV focuses mainly on the generation of SVA for SystemVerilog [22] designs [6], [11], [12].

Once the properties are written, FPV is conducted using a formal verification tool, where open-source variants like SymbiYosys [23] and commercial variants like Cadence JasperGold [24] exist. The proof of the property itself depends on the checker employed by the tool. Generally, a combination of bounded model checking [25] and k-induction [26] is used to guarantee an unbounded proof.

III. Proposed Modular End-to-End Pipeline

A. Overview

Figure 1 shows an overview of our proposed pipeline. As input, the pipeline receives a set of specification documents for a given design in textual form. As output, the pipeline provides complete FPV evaluation results, which can then be analyzed by a human expert. Our pipeline is divided into four stages, each with a distinct purpose. In the first stage, specification documents are pre-processed into an LLM-suitable format. In the second stage, assertions are generated with the help of a carefully engineered prompt. In the third stage, assertions are post-processed to align them with the RTL implementation. In the fourth stage, FPV evaluation is carried out directly as part of our pipeline. Within these four stages, the pipeline goes through several processing steps. Each of these processing steps is conducted with a custom prompt by an LLM and produces an output in *JavaScript Object Notation (JSON)* format, which is further transformed in subsequent steps. Note that all LLM calls follow the same flow as shown in the assertion generation stage in Figure 1, where a prompt is constructed from a prompt template and incoming data from a previous processing step is passed to the LLM. Also note that for all steps, the same LLM is used, which is chosen by the user. In the following, we go into detail on each of the four pipeline stages and outline how processing steps conduct their transformations.

B. Specification Pre-Processing

The purpose of the first pipeline stage is to break down complex specification documents by pre-processing. Especially for industry-scale hardware designs, the number and length of their specification documents can hinder the ability of LLMs to identify relevant properties and generate high quality assertions. Properties may also be specified in different parts of the specifications, or multiple properties may be hidden within a single sentence, as illustrated by Listing 1. In addition, specification documents may be written in different styles and for different target audiences, creating a divergence in

Listing 1. Example snippet of a natural language specification

```
Each FSM is disabled when `enable` is low.
Disabling the FSM is equivalent to an FSM reset, and both
     operations place the FSM in the `IDLE` state.
While in `IDLE`, the other state machine registers assume
     their default states:
The internal counters, the clock-divide, `bit_cnt` and
     `rep_cnt` all reset to 0, as does `pcl_int`.
```

assertion quality. To combat these issues, we transform all relevant specification documents for a given design into a set of properties in an *Intermediate Representation (IR)* format in three processing steps. In the first step, all specification documents are split into chunks, where coherent parts of the specification describing a certain functionality are extracted. The contents of the specification in these chunks remain unchanged, and they are mapped to the specification document from which they originate. As a next processing step, the extracted specification chunks are transformed into a structured functional description, keeping track of relevant signal names and identifiers given in the specification documents. Relevant functional information, such as affected registers, timing constraints, and clock domains, is also part of this functional IR. In the final specification pre-processing step, this functional IR is transformed to a property IR, which aims to match the structure of assertions to be generated next, while still remaining on the abstraction level of natural language. With this property IR, we ensure that complex specification documents are split into appropriately sized chunks and represented in a common style that is suitable for translation to SVAs in the next pipeline stage.

C. Assertion Generation

In the second pipeline stage, assertions are generated from the pre-processed specification. To this end, we employ the prompt template shown in Listing 2. In the beginning of the prompt, the context is established, and the general task is outlined. Next, the shape of the JSON input is defined, so the LLM can process it correctly. SVA construction rules guide the assertion generation by providing rules regarding different aspects, e.g., proper clock and reset handling. For the output format, the concrete JSON schema is defined, which contains an assertion object for each generated SVA. In addition to the SVA itself, this assertion object documents a status determined by the LLM, a potential error reason, and the natural language property to which the SVA maps. The global status indicates whether all assertion objects pass, some need further input or there are some errors. Instructions on this global status are given to the LLM after the output format definition. The failure path information describes how the LLM should behave in case of errors and notes provide further formatting requirements. Finally, the property IR produced in the specification pre-processing step is filled in the prompt template. The LLM generates assertions based on the prompt created from this prompt template for a given execution of the pipeline. Once assertions are generated, they are passed onto the next pipeline stage for assertion post-processing.

Listing 2. Prompt template for the generation of assertions

```
You are a senior SystemVerilog Assertion (SVA) engineer.
     Convert the given property specifications into VALID
     IEEE 1800 SVA that preserves intent and timing. Act
     deterministically in ONE pass and then STOP.
INPUT SHAPE:
- Input will be provided inline as JSON.
- The input may be:
  - a JSON array of items, OR
  - a JSON object containing an array at path
       "properties", OR
  - a JSON object containing an array at path
       "properties.properties".
...
SVA CONSTRUCTION RULES:
..
OUTPUT FORMAT (exactly one JSON object; no prose, no code
     fences, no extra text):
{{
  "assert_obj": [
    {{
      "status": "ok" | "needs_input" | "error",
      "error": "Non-empty reason if status!='ok'; else
           \"\"",
      "functionality": "Brief note of what is asserted and
           any clock/reset assumptions.",
      "nl_property": "If NL text was used, copy it; else a
           minimal summary from structured fields.",
      "sva": "One complete SVA (property + assert) with
           \\n escapes and a trailing \\n; MUST be '' if
           status!='ok'."
    }}
  ],
  "global_status": "ok" | "partial" | "error"
}}
GLOBAL STATUS:
...
FAILURE PATH (no loops):
...
NOTES:
...
INPUT (already substituted below):
<Property IR>
```

D. Assertion Post-Processing

The purpose of the third pipeline stage is to improve the quality of generated assertions by post-processing. The SVAs generated in the previous stage ideally correspond to the behavior specified in the property IR, but might not be aligned with the RTL implementation. To achieve this alignment and improve syntactic validity, we extract a list of signal names from the RTL design, to match it with the identifiers used in the generated assertions. It is important to note that no further implementation details are extracted from the RTL design, so that assertions are based purely on the specification and a bias to replicate the implemented functionality is avoided. The mapping step receives the extracted signal list, together with the assertions generated in the previous pipeline stage. In this first post-processing step, the assertions are rewritten to use only identifiers from the signal list. If there are multiple candidates for the signal name, the LLM should choose the most likely candidate and record a confidence score that it selected the correct one. In the resolving step, assertions with unresolved identifiers or a low confidence score below 0.7 are filtered out, to only evaluate SVAs which have a high probability to improve verification quality. As a final post-processing step, assertions are syntactically validated with a formal verification tool in the loop. To this end, each SVA is embedded in a minimal template and linted by the formal tool. For each provided error message, a feedback snippet is given

Listing 3. Example property before (top) and after (bottom) post-processing

```
property prop_done_ch0_event;
@(posedge clk_i) disable iff (!rst_ni)
    $rose(event_done_ch0_l) |-> ##1 (!event_done_ch0_l)
endproperty
assert property (prop_done_ch0_event);
property prop_done_ch1_event;
@(posedge clk_i) disable iff (!rst_ni)
    $rose(event_done_ch1_l) |-> ##1 (!event_done_ch1_l);
endproperty
assert property (prop_done_ch1_event);

property prop_done_ch_event_separation;
@(posedge clk) disable iff (!rst_n)
    ((done_ch0 && !$past(done_ch0)) |-> ##1 (!$rose(
        done_ch0))) && ((done_ch1 && !$past(done_ch1)) |->
        ##1 (!$rose(done_ch1)));
endproperty
assert property (prop_done_ch_event_separation);
```

to the LLM, with the task of fixing the detected syntax error. It is important to note that the LLM is instructed to fix only syntactic validity, without changing the semantics of the SVA. The number of iterations of this feedback loop by the formal verification tool can be configured by the user and defaults to one. An example of a property before and after post-processing is displayed in Listing 3. Note that not all generated properties will follow the same pattern, e.g., different properties may reason over different clock domains or reset states. After this assertion post-processing stage, we obtain a mapped, resolved, and linted set of assertions which are ready to be used for FPV evaluation in the final pipeline stage.

E. Formal Property Verification Evaluation

In the final pipeline stage, the processed assertions are evaluated and FPV is carried out. An important aspect of our end-to-end pipeline is the fact that this evaluation stage is directly integrated, i.e., no manual intervention is required. As a first step, all post-processed assertions are merged in an SVA block to allow simultaneous evaluation. This block is merged into a design-specific assertion module, which contains all relevant signal definitions for binding to the RTL implementation. Since we evaluate OpenTitan modules with our pipeline, we utilize the *dvsim* tool provided in the OpenTitan repository to perform FPV. As a formal verification tool, we employ Cadence JasperGold by default. Once FPV has been executed by dvsim, we process its output to generate an evaluation log, allowing for an easy analysis by a human expert. The format of our evaluation log is outlined in Listing 4. First, it lists metadata related to the FPV, including the name of the module, the path to the assertion module, and the path to the log generated by dvsim. Next, a summary of the FPV results is given, starting with the return code of dvsim and a task-specific name. Metrics on the generated SVAs include the number of errors, the number of successfully proven SVAs and the pass rate across all assertions. Regarding the coverage of the RTL design, formal, stimuli, and checker coverage evaluated by JasperGold are reported. To allow for an analysis of which SVAs are incorrect, failing, or unreachable, they are recorded. The summary window contains the raw output log given by the dvsim tool, in case further details are required. Finally, the evaluation log provides further metadata, including the

Listing 4. JSON format of the evaluation log generated by our pipeline

```
{
  "dut": <dut_name>,
  "assert_file": <assert_path>,
  "local_log": <log_path>,
  "summary": {
    "rc": <return code>,
    "name": <task name>,
    "errors": x,
    "warnings": x,
    "proven": x,
    "cex": x,
    "undetermined": x,
    "covered": x,
    "unreachable": x,
    "pass_rate": xx.xx,
    "cov_rate": xx.xx,
    "formal_coverage": xx.xx,
    "stimuli_coverage": xx.xx,
    "checker_coverage": xx.xx,
    "status": <pass/fail>,
    "failing_asserts": [],
    "unreachable_checks": [],
    "summary_window": ...,
  "cmd": <dvsim command>,
  "repo": <opentitan repo>,
  "meta": ...
}
```

specific dvsim command which was used and the path to the OpenTitan repository where it was executed. With the help of this evaluation log, a human expert can easily analyze the FPV evaluation results provided by the formal verification tool, which constitute the final result of our end-to-end pipeline.

F. Modularity

Another essential aspect of our pipeline is modularity. In the following, we highlight three design choices that are aimed at adding processing steps or making changes to our proposed pipeline as simple as possible. Firstly, our pipeline is modeled as a graph structure, where nodes represent individual processing steps, and edges represent the transitions between them. This allows individual processing steps to be implemented independently of each other, connecting them at a central location where the pipeline graph is constructed. Data transfer between nodes takes place through an internal state, which can be extended in case further information needs to be exchanged. Another feature of the graph representation are conditional branches, which can be employed to skip processing steps on a given condition, e.g., evaluation is skipped if no linted assertions can be generated. With this graph structure, we enable the simple addition of further pipeline nodes and modification of the pipeline flow. Secondly, prompt templates are designed to accommodate for this flexibility, making minimal assumptions on the input shape and filling in input data based on the previous processing step. As can be seen in Listing 2, the input JSON schema is not specifically enforced, only listing optional input shapes. The input data itself, in our case the property IR, is inserted at the end of the prompt template, such that the LLM can automatically infer its shape and process it independently of the output of the previous pipeline node. Prompt templates are additionally kept separate from the source code, making sure that no interference with processing step modifications occurs. Finally, the evaluation step is kept in its own pipeline node, making

TABLE I
EXPERIMENTAL RESULTS OF FPV FOR TWO OPENTITAN IP DESIGNS USING OUR PIPELINE VS. A ZERO-SHOT BASELINE

OpenTitan IP	LLM	RT [m]	API Cost [$]	# SVA	Syntax Pass %	FPV Pass %	Assertion %	Formal %
aon_timer	Gemma 3	115	0.00	158	35.4%	27.1%	92.1%	8.3%
	Qwen 3	143	0.00	117	61.5%	25.0%	87.8%	40.2%
	GPT-5 mini	130	1.58	**222**	66.2%	**49.7%**	**95.2%**	**67.6%**
	Baseline	2	0.02	27	**100%**	24.1%	90.9%	36.3%
pattgen	Gemma 3	129	0.00	196	17.9%	20.0%	95.5%	0.9%
	Qwen3	186	0.00	98	78.6%	33.7%	83.0%	18.0%
	GPT-5 mini	148	1.59	**230**	23.9%	**94.7%**	94.6%	**38.4%**
	Baseline	2	0.01	65	**96.9%**	49.2%	**100%**	31.4%

#SVA: Number of generated assertions **RT [m]**: Runtime in minutes rounded down **API Cost [$]**: Cost for run in $ **Syntax Pass %**: Percentage of assertions with valid syntax **FPV Pass %**: Percentage of assertions satisfied by JasperGold **Assertion %**: Assertion coverage given by JasperGold **Formal %**: Formal coverage given by JasperGold

sure that it can be adjusted according to the design to be verified. As we consider OpenTitan IP in the current version of the pipeline, evaluation is offloaded to the provided dvsim tool. However, in the general case, an alternative evaluation step may be implemented, giving full control over the FPV depending on the RTL design to be verified. Together, these three aspects ensure that our end-to-end pipeline is modular, offering opportunities for future extension and modification.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

To evaluate the syntactic and semantic quality of assertions generated with our pipeline, we implement it in Python [27]. We execute our pipeline on two OpenTitan IP designs, which did not previously support FPV at the time of writing: *Alywas-On Timer (aon_timer)* and *Pattern Generator (pattgen)*. For both designs, we conduct pipeline runs with three different LLMs to observe the influence of model choice on assertion quality. In addition, we generate assertions with a zero-shot prompt baseline without intermediate processing steps to evaluate if our pipeline is able to achieve any improvements. We omit a comparison to previous works due to the lack of common evaluation metrics. Pipeline runs utilize Google DeepMind's open-weight Gemma 3 [28] with 27 billion parameters, Alibaba's open-weight Qwen 3 [29] with 30 billion parameters, and OpenAI's closed-weight GPT-5 mini [30]. For the baseline, OpenAI's closed-weight GPT-5 Chat [31] is used. Open-weight models are run locally on a server with an Nvidia A40 data center graphics card with 48 gigabytes of memory. The smaller GPT-5 mini is called via the API and the larger GPT-5 Chat is accessed through the ChatGPT [32] interface.

B. Experimental Results

Table I shows the experimental results of the evaluation of FPV for two OpenTitan IP designs using our pipeline against the zero-shot baseline. For each execution, runtimes in minutes and API costs are reported. As a measure of model verbosity, we track the total number of assertions generated by each run. For both designs, our pipeline generates a significantly higher number of SVAs, with up to 8.2x more assertions generated by GPT-5 mini compared to the baseline for aon_timer. To evaluate syntactic quality, we consider the percentage of assertions that pass syntax checks conducted by JasperGold. Here, the baseline scores very well, with 100% for aon_timer and almost 97% for pattgen. Regarding our pipeline, GPT-5 mini scores highest for aon_timer, while Qwen 3 scores highest overall with 78.8% for pattgen. Generally, syntax pass percentage stays below 80% for our pipeline runs. For the evaluation of semantic quality, we look at a total of three metrics. Note that these metrics only consider those assertions which previously passed the relevant syntax checks. FPV pass observes what percentage of assertions pass for the design, excluding unsatisfiable or uncovered SVAs. Across both designs, our pipeline with GPT-5 mini is able to achieve the best FPV pass rate, with up to almost 95% for pattgen and scoring more than two times higher than the baseline for aon_timer. Gemma 3 and Qwen 3 perform similarly on both IP designs, slightly outperforming the baseline for aon_timer and lacking behind for pattgen. Assertion coverage indicates what percentage of SVAs affect the design and are not covered vacuously. This metric is high across all runs, with all models except for Qwen 3 consistently scoring above 90%. For aon_timer, our pipeline once again achieves the best performance, while for pattgen it is not able to reach 100% like the baseline does. Formal coverage, in contrast to assertion coverage, expresses what percentage of the design is covered by the given set of properties. Here, GPT-5 mini once again scores highest across both designs, reaching 67.6% formal coverage and thus 86% more than the baseline for aon_timer. Notably, Gemma 3 performs very poorly in this metric, staying below 10% across both designs, while Qwen 3 outperforms the baseline at least for aon_timer. Overall, three trends can be observed. Firstly, the syntactic quality of assertions generated by our pipeline can not reach to the very high syntax pass percentage of assertions generated by the baseline. A possible explanation for this might be the much higher number of assertions generated by our pipeline, resulting in more variance of syntactic quality. Secondly, our pipeline is able to improve semantic quality compared to the baseline across all three metrics we consider. While the assertion coverage of SVAs generated by the baseline is very competitive and even reaches 100% for pattgen, our pipeline is able to achieve much higher scores across FPV pass percentage and formal coverage percentage, outperforming the baseline by more than 2x and 86% respectively. Thirdly, our evaluations show that the model choice greatly influences the syntactic and semantic quality of assertions generated using our pipeline. The open-weight models Gemma 3 and Qwen 3 generally scored very poorly across all metrics, only offering some exceptions, such as the high syntax pass rate achieved by Qwen 3 for pattgen. However, it is important to note that,

although exact parameter counts are not disclosed for GPT-5 Chat, these two models are very likely much smaller in scale than the baseline, which could point to a possible explanation. GPT-5 mini as a closed-weight model on the other hand shows promising result with the integration into our pipeline, despite the supposedly smaller model size compared to the baseline. At the same time, our evaluation still shows room for improvement in metrics like syntax pass percentage and assertion coverage percentage, hinting that further investigation into our pipeline and the model choice could prove valuable.

V. Conclusion & Future Work

In this work, we introduced a modular, end-to-end pipeline for FPV using LLMs. We showed how our four pipeline stages, specification pre-processing, assertion generation, assertion post-processing, and FPV evaluation, can be implemented through several processing steps, transforming a set of specification documents into comprehensive FPV results ready for human analysis end-to-end. In addition, we described how modularity was considered as a central aspect of our pipeline, enabling the extension and refinement of further processing steps. Experimental evaluation of our proposed pipeline against a zero-shot baseline showed how the quality of generated assertions can be improved and that model choice matters in this regard. Future work may focus on integrating a broader set of designs, both within and independently of OpenTitan. To gain a deeper understanding of the impact of individual pipeline steps, ablation analysis may be conducted. Furthermore, the integration of additional processing steps into individual pipeline stages can be evaluated. For example, approaches like RAG or self-refinement explored in related works could be integrated into our modular pipeline.

Acknowledgements

This work was supported by the German Ministry for Research, Technology and Space (BMFTR) with project *ExaVerse* (grant number FKZ 16IW25003).

References

[1] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[2] R. Drechsler, *Advanced formal verification*. Boston: Kluwer Academic Publishers, 2004.

[3] P. DasGupta, *A Roadmap for Formal Property Verification*, p. 217–241. Dordrecht: Springer Netherlands, 2006.

[4] L. Fix, *Fifteen Years of Formal Property Verification in Intel*, vol. 5000 of *Lecture Notes in Computer Science*, p. 139–144. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[5] A. Yadav, A. Patel, and M. Shah, "A comprehensive review on resolving ambiguities in natural language processing," *AI Open*, vol. 2, 2021.

[6] 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), IEEE, 2025.

[7] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, and R. Karri, "ChIRAAG: ChatGPT informed rapid and automated assertion generation," in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, (Knoxville, TN, USA), p. 680–683, IEEE, 2024.

[8] S. Paria, A. Dasgupta, D. R. Ankireddy, P. Chakraborty, and S. Bhunia, "FV-PAL: Scalable formal verification through partitioning and LLM-guided property generation," in *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, pp. 768–775, 2025.

[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, p. 4374–4389, 2024.

[10] S. Paul, A. Banerjee, S. Ghosh, S. Surendran, and R. K. Gajavelly, "LISA: Llm informed systemverilog assertion generation with rag and chain-of-thought," in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, vol. 1, 2025.

[11] Z. Yan, W. Fang, M. Li, M. Li, S. Liu, Z. Xie, and H. Zhang, "AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, (Tokyo Japan), p. 614–621, ACM, Jan. 2025.

[12] A. Gupta, B. Mali, and C. Karfa, "SANGAM: Systemverilog assertion generation via monte carlo tree self-refine," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pp. 180–187, IEEE, 2025.

[13] S. Paul, A. Banerjee, S. Ghosh, S. Surendran, and R. K. Gajavelly, "SuperSAGA: A supervisor-subordinate agentic workflow for the generation of assertions," in *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 420–425, 2026.

[14] H. Lyu, Y. Wang, J. Zhou, Z. Chao, T. Wang, and H. Li, "AssertMiner: Module-level spec generation and assertion mining using static analysis guided llms," 2025.

[15] H. A. Quddus, S. Hossain, and Z. Cevahir, *Enhanced VLSI Assertion Generation: Conforming to High-Level Specifications and Reducing LLM Hallucinations with RAG*. DE: VDE VERLAG GMBH, Nov. 2024.

[16] lowRISC contributors, "OpenTitan." https://opentitan.org/, 2026. accessed 2026-03-18.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[18] K. Xu, D. Schwachhofer, J. Blocklove, I. Polian, P. Domanski, D. Pflüger, S. Garg, R. Karri, O. Sinanoglu, J. Knechtel, Z. Zhao, U. Schlichtmann, and B. Li, "Large language models (LLMs) for electronic design automation (EDA) : Special session paper," in *2025 IEEE 38th International System-on-Chip Conference (SOCC)*, 2025.

[19] M. Abdollahi, S. F. Yeganli, M. A. Baharloo, and A. Baniasadi, "Hardware design and verification with large language models: A scoping review, challenges, and open issues," *Electronics*, vol. 14, p. 120, Dec. 2024.

[20] M. Hassan, M. Nadeem, K. Qayyum, C. K. Jha, and R. Drechsler, "Prompt. verify. repeat. LLMs in the hardware verification cycle," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 2025.

[21] K. Qayyum, S. Ahmadi-Pour, C. K. Jha, M. Hassan, and R. Drechsler, "LLMs for hardware verification: Frameworks, techniques, and future directions," in *2024 IEEE 33rd Asian Test Symposium (ATS)*, 2024.

[22] IEEE Computer Society, *1800-2023 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. Institute of Electrical and Electronics Engineers, 2024.

[23] YosysHQ, "SymbiYosys." https://yosyshq.readthedocs.io/projects/sby/en/latest/, 2023. accessed 2026-03-17.

[24] Cadence, "JasperGold." https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, 2026. accessed 2026-03-17.

[25] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking.," *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[26] T. Wahl, "The k-induction principle," *Northeastern University, College of Computer and Information Science*, pp. 1–2, 2013.

[27] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[28] Google DeepMind, "Gemma 3." https://deepmind.google/models/gemma/gemma-3/. accessed 26-04-11.

[29] Alibaba Cloud, "Qwen 3." https://github.com/QwenLM/Qwen3. accessed 26-04-11.

[30] OpenAI, "GPT-5 mini Model." https://developers.openai.com/api/docs/models/gpt-5-mini. accessed 26-04-11.

[31] OpenAI, "GPT-5 Chat Model." https://developers.openai.com/api/docs/models/gpt-5-chat-latest. accessed 26-04-11.

[32] OpenAI, "ChatGPT." https://chatgpt.com/. accessed 26-04-11.