GenAI Based Assertion Code Pattern Generation

Mrinal Kanti Kar, Yash Kumar, Rohit Kumar, Karthikeyan Sundaram, Sathish Kumar K, Manikandan S, Murthy Siriginedi NVS, and Sharanbasappa

HCL Technologies, India

E-mail: { mrinalkantik, yash.kumar, rohitku, karthikeyan.sundara, sathishkumar.kri, s-manika, smurthy, sharanb }@hcltech.com

Abstract **- Newer designs with increasing complexities have significantly increased the verification effort required to validate them. To prove a design meets its design specification, assertion-based verification is one popular verification technique used widely due to its dynamic (simulation based) and static (Formal Verification) nature. However, coding these assertions efficiently and implementing them accurately to meet the design specification is a challenging task and a time-consuming human effort. Automating the process of generating the assertion codes directly from the design specification could potentially reduce the overhead of time and human effort as well as avoids unintentional errors in coding these assertions. A recent advancement in the field of Generative Artificial Intelligence (GenAI) is Large Language Models (LLM), a promising approach from the proliferation of technically capable code-writing by comprehending natural language inputs. This aspect encouraged us to investigate using an LLM to automatically generate the assertion codes by feeding them with design specification directly. Further, instead of generating individual assertions, this paper details the approach of generating assertion patterns which can then be used as a library to generate the required assertion instances.**

I. INTRODUCTION

A. Background: Assertion-Based Verification

Assertions are invaluable part of Design Verification (DV) activity. They are used to catch illegal behavior Of Design Under Test (DUT) in simulation-based verification (dynamic verification) and are also central to Formal verification (static checking). In case of Formal Verification, assertions serve as description of design properties. These design properties must not be violated by the DUT. Typically, assertion intent is captured in the test plan (both in Formal and simulation). These are normally described in general English language which are subsequently implemented using System Verilog Assertions (SVA). This implementation of assertions using SVA is a typically very time-consuming human coding effort. One of the important points to note is that these assertions can be easily expressed in plain English language, unlike a lot of other technical specifications which need certain domain specific languages (e.g. UML) to express the intent.

Simulation-based verification enables verification in a virtual simulation environment closer to the real environment in which the systems will work. Based on the inputs, it executes meaningful and possible scenario planned to verify that the system complies with the design specifications. It is very much scalable if modeled properly and provides re-usability. Assertions can be embedded or bound to the design and are automatically checked during the simulation.

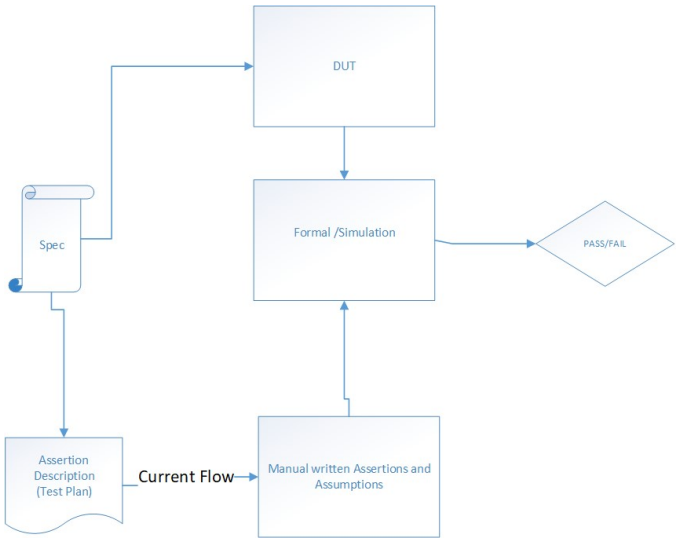

Figure 1. Assertion-Based Verification – Current/Existing Flow.

Formal verification is rather a mathematical approach in verifying the correctness of the designs. It is in way an automated testing method that detects many common design errors in their early stage. Unlike simulations, it provides exhaustive checking of all possible paths and provides a comprehensive result with no overhead of need for a Testbench environment. SVA is an integral part of Formal verification and plays a crucial role in proving the design behavior. The SVA can be used directly in Formal verification and the design failures can be easily detected with the assertion failure and Formal tool helps in coverage of exhaustive state space of the design. Hence, Formal verification is gaining popularity and widespread adoption.

One of the recent advancements in the field of AI has been the rise of Generative AI, and one of its promising approaches comes from the proliferation of technically capable code-writing large language models (codeLLMs). In the realm of coding, LLMs demonstrate a distinct and inventive method. Beyond comprehending natural language, they undergo training on vast repositories of code, encompassing single programming languages or a fusion of them. These training datasets can amass hundreds of gigabytes. The prompts used for these code-trained LLMs can take various forms, including instructions, comments, code snippets, or a combination of these.

In this paper, we describe how we used the chat GPT (a popular LLM) to experiment with SVA code generation and the results we got from this work.

II. DESCRIPTION

B. Challenges: Assertion-based Verification

1. Assertion coding/implementation is a time-consuming manual effort, particularly for complex designs.
2. Assertion implementation requires skills to understand the design intent and convert it to an assertion code.

3. Assertions must be defined carefully to represent the design intent accurately.
    a. Incorrect assertions can give misleading results. i.e. false positives/false negatives
    b. Debugging an assertion can be difficult if design intent is misinterpreted.
    c. How do we know whether enough assertions have been written to cover all valid and corner case scenarios?
    d. Customization and flexibility to cater design changes are very minimum and needs additional efforts.

C. Challenges: Script-based automation for SVA code generation

A script is nothing but a set/sequence of instructions/commands that can be executed by a computer or programming language. It helps in automating some of the processes that would otherwise have to be performed manually involving human effort. There have been a lot of prior work done towards a script-based automation to the process of assertion code generation, but they do have their own challenges and limitations.

1. Developing/customizing the scripts take tremendous effort and very much time-consuming. Particularly in complex designs.
2. Thorough planning is required as any minor changes in design would cost more time.
3. Most of the time, less re-usable for a different design/environment and might need rework and effort.
4. Sometimes needs a lot of input as well as intermediate files to execute.

D. Proposed Solution

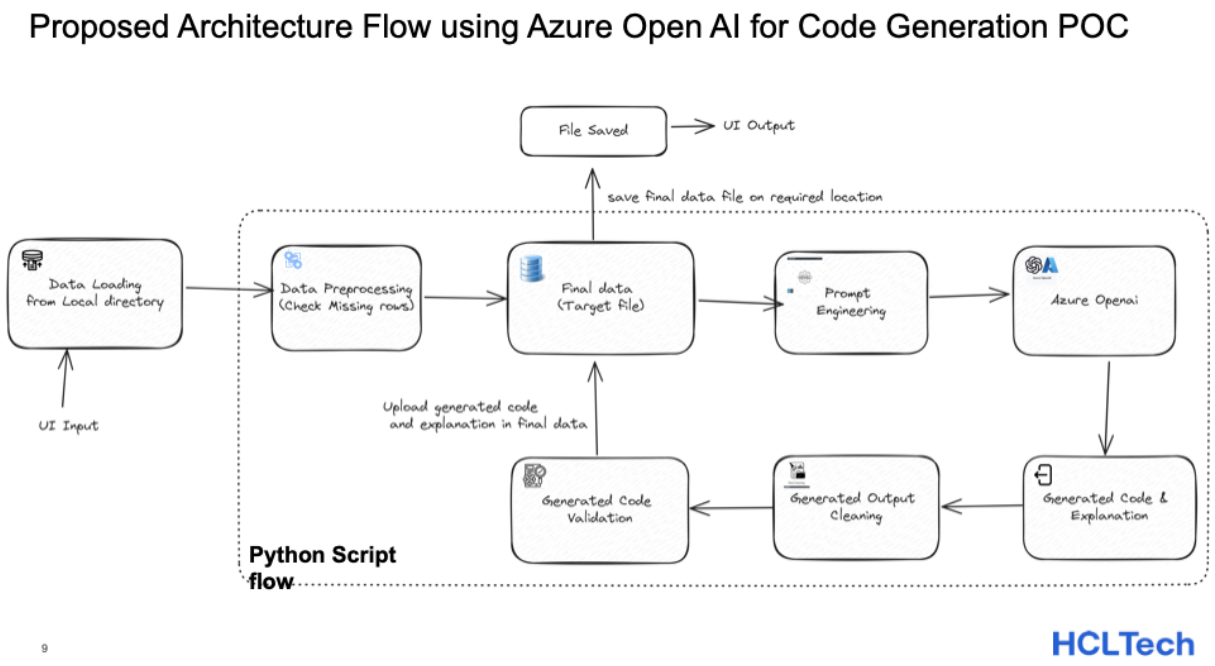

Figure 2. Flow Diagram - Proposed GenAI-based Solution

We adopted a strategic approach to address the key challenges in SVA code generation. Our approach was to utilize the advanced natural language processing and code generating capabilities of OpenAI GPT-

3.5 turbo model to generate assertion code based on predefined descriptions. We wanted to develop a chatbot interface for seamless interaction and SVA code generation, along with collaboration with system Verilog experts to validate the accuracy and syntax of the generated assertion code.

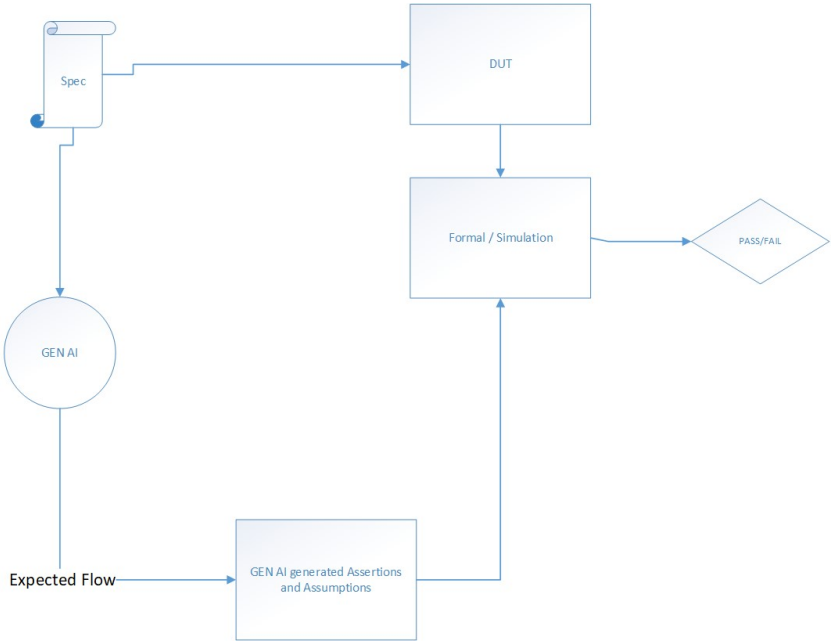

Figure 3. Assertion-Based Verification – Proposed/Expected flow.

D. Novelty of the Solution

There are a couple of papers available in public domain which have done similar work.

1. LLM-assisted Generation of Hardware Assertions (Reference [1])
2. ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation (Reference [2])

The salient features of the [1] are as follows,

a. The use-case considered are assertions for security features.
b. Uses very brief, brief and detailed description of the assertion intent approach for their evaluation.
c. Uses open source OpenTitan database for the exercise

The salient features of the [2] are as follows,

a. Uses open source OpenTitan database
b. A tool is used to digest the design information and feed to the LLM.

We take a different approach from the approach taken by the references [1] and [2] listed above. Based on experience in using assertions in multiple projects, we noticed that invariably, the entire assertion instances can be mapped to a small set of assertion patterns. So, instead of generating the full assertion list, we took the approach of generating the assertion patterns which can then be used as

actual instances.

The added advantage of the solution is that it would eliminate the need for preparation of any input/intermediate files such as assertion specification/intent, input constraint files, templates etc. that would cost time and increase the design verification cycle significantly.

E. Application

Our work can be used in 2 ways in case of Silicon design work, as follows,

1. Create assertion patterns for SV/UVM based verification.
2. Create assertion patterns for formal property verification.

In both the cases, our expectation is that as we refine our approach and with a rich data set, the human effort to code the assertions will continue to reduce.

F. SVA Output Validation

We obtained a sample of open-sourced specifications and passed them one by one through the OpenAI GPT-3.5-turbo Model. We understood key aspects involved in generating code assertions from descriptions and engineered prompts to generate SVA codes. The results were analyzed, followed by iterative prompt re-engineering applying Chain-of-Thought prompting and few-shot learning to enhance the quality of generated codes.

Provided below is a generated code result, where Input is the natural language low-level design specification, and Output is code snippet generated from OpenAI GPT-3.5-turbo:

2.2.2 Write Operation:

The Setup phase of the write transfer occurs when the select signal, PSEL, is asserted, which means that PADDR, PWRITE, and PWDATA must be valid. The Access phase of the write transfer occurs when PENABLE is asserted. PREADY is asserted by the Completer at the rising edge of PCLK to indicate that the write data will be accepted. PADDR, PWDATA, and any other control signals must be stable until the transfer is complete. At the end of the transfer, PENABLE is de asserted. PSEL is also de asserted unless there is another transfer to the same completer. The PWDATA is determined based on PADDR, If PADDR matches with any of the output registers of the APB, the PWDATA will be written to the corresponding APB output registers based on the provided address.

Figure 3. Specification fed as an Input to GenAI

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// 2) PADDR, PWDATA and PWRITE are stable and defined during the transfers.

// Check that PADDR and PWRITE are stable during the cycle
property APB_ADDR_WRITE_STABLE;
  @(posedge PCLK) (PSEL |-> $stable({PWRITE, PADDR}));
endproperty

// Check that PWDATA is stable during the cycle
property APB_WRITE_DATA_STABLE;
  @(posedge PCLK) ((PSEL && PWRITE) |-> $stable(PWDATA));
endproperty

// If PENABLE==TRUE, it must be in the second clock of a cycle, and then it must be turned off
property APB_NO_PENABLE_OUTSIDE_CYCLE_2;
  @(posedge PCLK) (PENABLE |-> $stable(PSEL) ##1 (!PENABLE));
endproperty

assume_APB_WRITE_DATA_STABLE : assume property(APB_WRITE_DATA_STABLE);
assume_APB_ADDR_WRITE_STABLE : assume property(APB_ADDR_WRITE_STABLE);
assume_APB_NO_PENABLE_OUTSIDE_CYCLE_2 : assume property(APB_NO_PENABLE_OUTSIDE_CYCLE_2);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Figure 4. Output assertion code from GenAI

The sample SVA assertion codes output from the GenAI was compiled/validated using a Formal verification tool and they are passing the static checks currently as shown in the snippet below:

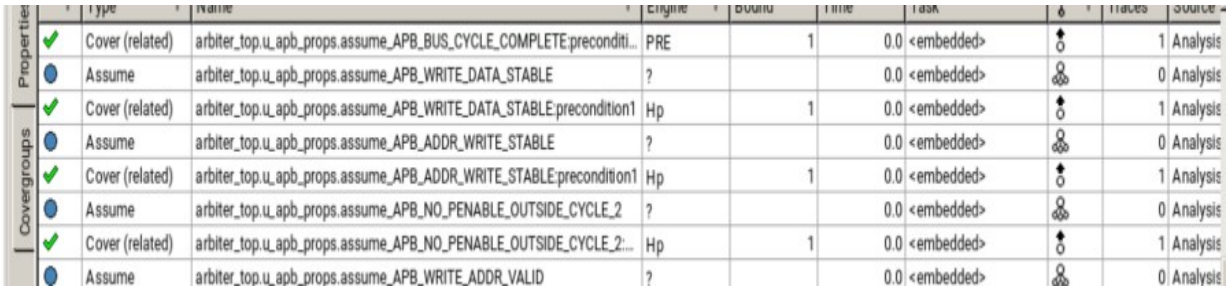

Figure 5. Formal verification results

G. *Benchmarks*

Currently, a good mix of open source as well as application specification protocol/design specifications were tried with the proposed solution of GenAI-Based assertion codes and below table shows the accuracy of the code generation obtained for each of their corresponding specification inputs. We are continuously working on using varied inputs and improving the output accuracy.

Specification Input to GenAI	**Manual written Assertion accuracy**	**Gen AI Assertion**
APB Protocol	100%	90%
AHB Protocol	100%	75~80%
AXI Protocol	100%	65~70%
Communication Based Design	100%	around 70%
Image Based Design	100%	around 70%

Table 1. Initial Benchmark results

III. FUTURE WORK

One of the key challenges in SVA code generation is quantifying LLM model performance through appropriate evaluating metrics. While datasets like HumanEval exist for software, large-scale test problems and methods to evaluate the syntactical and functional correctness of LLM-generated HDL are lacking. As a part of the on-going work, we plan to carry out code generation on different varieties of assertion descriptions, and evaluate prompting technique, along with generated code quality using appropriate evaluating metrics. Another key aspect of our solution is data privacy and security, whereby we need to ensure client specific data does not get exposed to OpenAI through their model consumption. We are developing an Azure OpenAI service-based application to avoid any form of data leakage or exposure to OpenAI through their model consumption for more code generation, thereby ensuring responsible data usage best-practices. Model fine-tuning with SVA customized data is another area of focus for SVA generated code quality improvement.

IV. CONCLUSION

In conclusion, we do feel that GenAI has a significant role to play in generation of assertion code that can be deployed for simulation-based or Formal verification. It should also be noted that formal verification is taking a lot of space that was earlier taken by simulation methods. It is well known that, in Formal verification, a significant manual effort is spent in coding the assertions. Hence, we feel automation of assertion code generation will go a long way in improving the overall productivity.

ACKNOWLEDGMENT

We would like to extend our sincere thanks to all the members of IP/SoC verification teams who would take time to review this paper and provide us with their valuable feedback.

REFERENCES

[1] R. Kande et al., "(Security) Assertions by Large Language Models," in IEEE Transactions on Information Forensics and Security, vol. 19, pp. 4374-4389, 2024, doi: 10.1109/TIFS.2024.3372809.

[2] Bhabesh Mali et al., “ChIRAAG: ChatGPT Informed Rapid and Automated Assertion Generation,” in IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2024

[3] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper c. 2010 SystemVerilog Assertions Handbook, 4th edition for Dynamic and Formal Verification, ISBN 878-0-9705394-8-7. http://systemverilog.us/