GenAI Leap in Formal Verification Testplanning

Anshul Jain, Karan Rawat, Pradip Prajapati
anshul.jain@intel.com, karan.rawat@intel.com, pradip.prajapati@intel.com
Intel Corporation

***Abstract*- The complexity of modern hardware design necessitates robust verification methodologies. Formal Verification stands out for its rigor but is hampered by the intricate process of creating effective testplans. This paper introduces an innovative approach of integrating generative artificial intelligence into the process of formal property verification for testplanning. Our approach leverages prompt engineering for optimizing interactions with generative AI models to extract accurate formal verification testplan. We share a novel framework to develop high-quality prompts for formal property verification testplanning. In this work, we performed detailed experiments with state-of-the-art mainstream large multi-modal models and the results indicate a marked improvement in showcasing a reduction in effort required to create formal verification testplans from scratch. Our evaluation encompassing three different designs of variable complexity demonstrates a minimum of 80% accuracy in generating formal verification testplan. This paper not only bridges a gap in existing literature but also opens new avenues for research in applying prompt engineering to formal verification process, potentially revolutionizing practices in formal verification sign-off.**

***Keywords*- Formal Verification (FV), Testplanning, Generative AI, Prompt Engineering, Large Language Models (LLMs), Large Multi-modal Models (LMMs), Formal Property Verification (FPV).**

I. Introduction

In the semiconductor industry, the verification of hardware design is paramount to ensuring success of silicon systems. Functional verification is a vital part of the chip design process, focusing on ensuring that a design implementation meets its specified requirements. Initially, the architects draft these specifications in a natural language document. Subsequently, RTL designers convert the specifications into RTL implementation and verification engineers take on the task of validating that RTL implementation function as per the given specifications.

As technology scales to new heights with increasingly complex designs, need for robust verification has become becomes more critical and time consuming. Functional verification account for a significant portion of the development cycle and resources in chip design process. Traditional hardware design verification faces numerous challenges, including handling of ever-growing operations flows, ensuring comprehensive test coverage, and managing deep corner-cases scenarios of modern circuits.

Formal Verification stands out as a critical technique in this landscape. FV involves mathematically proving that design adheres to specified properties or requirements, offering a more rigorous and exhaustive approach compared to traditional simulation-based methods. Successful verification starts with a comprehensive testplan, and formal verification is no different. The effectiveness of formal property verification is heavily dependent on the quality and comprehensiveness of the properties planned for verification, a task that requires significant expertise and time.

With the recent advances in Generative AI, a question arises – "Can GenAI models help accelerate formal property verification testplan creation?" Recently, researchers have explored using GenAI models for code generation including SVA [16] generation. In this paper, we take an innovative approach to enhance the process of formal verification to generate formal verification testplan from design specification using prompt engineering. Prompt engineering involves crafting inputs (prompts) to a GenAI model to elicit desired outputs or behaviors.

Our approach: We aim to establish a robust framework for developing effectiveness prompt(s) to generating high-quality testplans using GenAI models. We aim to apply the concepts of prompt engineering for building prompt(s), use robust prompt evaluation criteria to refine the prompts for greater accuracy, thus streamlining the process of deploying GenAI in testplan creation. This approach has the potential to reduce the time and expertise required to effectively use FPV, thereby addressing some of the critical challenges in hardware design verification.

Our technical contribution: The primary objective of this paper is to provide the methodology for integrating GenAI in the process of formal verification testplanning using Spec-to-Testplan Prompt Engineering Cycle (shown in figure 1) to gain real productivity boost to the overall verification process. Proposed methodology will help fellow professionals in adopting GenAI capabilities to formal verification and open new avenues for research and application. This paper also provides a suggested sequence of prompts for generating testplans that has been tested to generate more than 80% accurate and exhaustive testplans. Our work includes detailed analysis of the methodology which was put to test in three case studies.

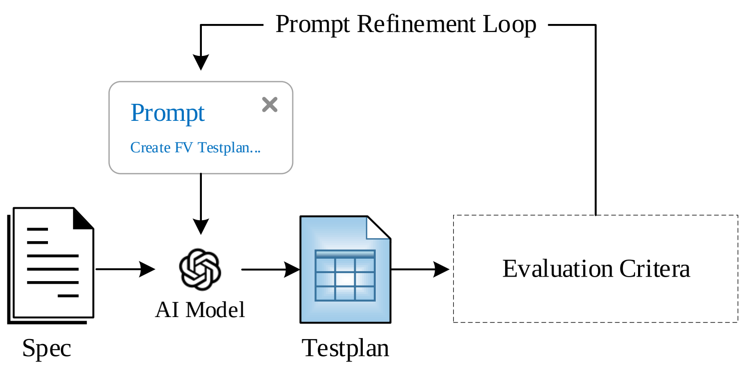

Figure 1: Spec-to-Testplan Prompt Engineering Cycle

II. BACKGROUND & RELATED WORK

A. *Formal Property Verification: Method & Challenges*

Formal Property Verification [15] has gained immense traction in digital design verification process. It allows verification engineers to encode their design specifications as assertions that can be validated against the implementation through model checking. A common approach to developing these assertions is through temporal logic, with SystemVerilog Assertions (SVAs). By ensuring that these assertions are satisfied, we can detect and address logic bugs early in the design cycle, improving the overall reliability of the product. Despite its effectiveness, one of the major challenges in FPV is the creation of comprehensive and high-quality assertions. The manual crafting of SVAs is time-consuming, requiring significant human effort and expertise.

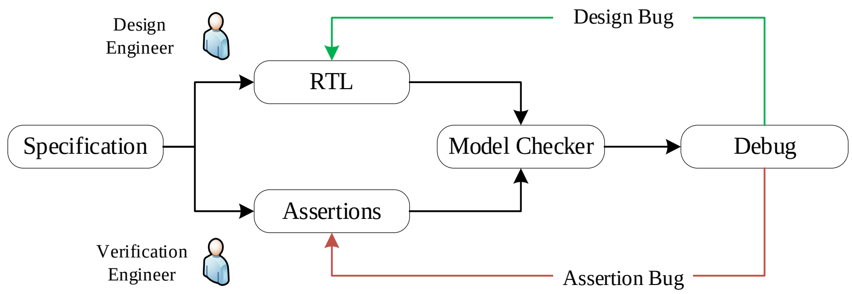

Figure 2: Formal Property Verification Flow

B. Automatic Assertion Generation: Methods & Limitations

To address the challenge of writing comprehensive and quality assertions, researchers and industry has focused on generating SVAs automatically. The automatic approaches can be categorized into 3 categories explained in table 1.

Table 1: Existing Techniques for Generating Automatic Assertions

Name	Method	Limitations
Static Analysis [1][2][3][4][5]	Conduct static analysis of the source code to infer potential properties and invariants, which can then be translated into SVAs.	Quality of generated assertions is good in terms of accuracy; however, the scope of assertions is limited to micro-architectural correctness and might not capture high-level design intentions.
		Static analysis can generate false positives or miss certain behaviors due to shallow COI of generated properties.
Dynamic Analysis [6][7][8]	Monitor the behavior of the design during simulation, conduct dynamic analysis to identify recurring patterns and behaviors that can be formalized as SVAs.	Generate assertions solely based on the same RTL design, without referencing a golden model. This approach can result in the creation of incorrect SVAs with flaws like those in RTL implementation.
		Quality of generated assertions is a function of operational modes and scenarios exercised in dynamic simulation. Therefore, generated assertions might not cover all scenarios leading to false failures.
ML-based Techniques [9][10][11][12][13]	Train machine learning algorithms to recognize common patterns in the design and automatically generate assertions for these patterns. Techniques like reinforcement learning can be used to explore the design space and maximize coverage.	Requires a large amount of data to train the models effectively. Machine learning models might not generalize well to unseen design patterns, leading to incomplete or incorrect assertions.
		Translating natural language into assertions is a highly complex task, requires deep understanding of the design functionality and specialized expertise in SVA. Therefore, accuracy of generated assertions is often low.

C. Generative AI Models for Hardware Design & Verification: Summary & Research Gaps

Recent advancements in Generative AI models like ChatGPT [14] have shown extraordinary capabilities in content generation and have evolved to assist humans in various roles. The application of these models in hardware design and verification is an emerging area of exploration that promises to revolutionize the field. These models are being utilized for tasks such as RTL code generation [17], generate and review specification documents [18], assertion generation [19], demonstrating their potential to automate and enhance design and verification processes. These diverse applications and ongoing research efforts underscore the potential of these models to significantly improve various aspects of chip design.

Research so far has emphasized on either training generative models from scratch or fine-tuning existing foundation models with specific datasets. While some these models are effective in generating assertions from the natural language specifications, none succeeded in generating both – accurate and exhaustive set of assertions for a design. We believe **decomposition of spec-to-assertion task into sub-tasks such as spec-to-testplan and testplan-to-assertion** will help bridge the gap in accuracy and exhaustivity.

In this paper, we focus on improving spec-to-testplan process with the aim to develop a methodology that achieves higher accuracy and exhaustivity than existing traditional and generative artificial intelligence techniques summarized in figure 3.

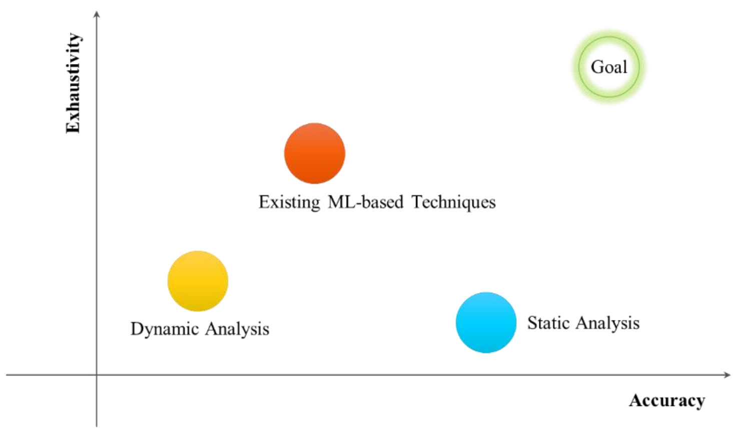

Figure 3: Comparative Evaluation of Assertion Generation Techniques

III. Problem Statement

Formal Verification Testplan is largely derived from the architectural/micro-architectural specification documents. These specification documents contain a vast number of unstructured details in natural language (mostly) as well as images (block diagrams, timing diagrams, stick diagrams, etc.) which can overwhelm human cognition. Cognitive psychology [20][21] suggests that human brain processes information more effectively when presented in structured formats like list. FV engineers too struggle absorbing big, chunky, unstructured details of the design under test (DUT) and arrange them into a structured and complete testplan.

Recently, generative artificial intelligence models have demonstrated remarkable capabilities in text understanding and generation, allowing for generalization across various domains and tasks [22]. Therefore, numerous tasks have been proposed to harness abilities of these models and offer solutions for Information Extraction (IE) tasks based on a generative paradigm. IE convert plain text into structured knowledge (e.g., relations, events, entities) and can serve the task of creating structured testplans from unstructured details in specification documents.

Our early experiments with GPT4 showed us that it could generate fairly structure testplans from specification document, but they were generic and partially non-actionable. Moreover, we found that the generated testplans were aligned towards simulation-based verification technologies. Our experiments also highlighted the reproducibility challenge with generation models because of their creativity – a key attribute of most of such models. One of the solution to these problems is to train a new model from scratch. Another solution is to fine-tune an existing foundation model like LLaMA [23] for creating formal verification testplans. Both solutions have limitations such as non-availability of quality formal verification testplans and huge cost of training/fine-tuning the models.

To address these challenges, we propose to tap into unexplored, low-cost, and readily applicable technique of using existing foundation models with intelligent use of prompt engineering principles to obtain desired results (formal verification testplans) in predictable fashion (pre-defined structure).

III. Proposed Methodology

Our methodology for extracting high-quality formal verification testplans from specification documents is based on three pillars – (1) Prompt Refinement Framework, (2) Prompt Building Mechanism and (3) Prompt Evaluations Metric. In the following text, we will share the details of each pillar and their interplay to create our proposed methodology.

A. *Prompt Refinement Framework*

We propose a comprehensive framework shown in figure 4 that can evaluate the testplans created by generative models objectively and provide suggestions for refining the prompts to arrive at robust, high-quality prompt script for a diverse variety of designs.

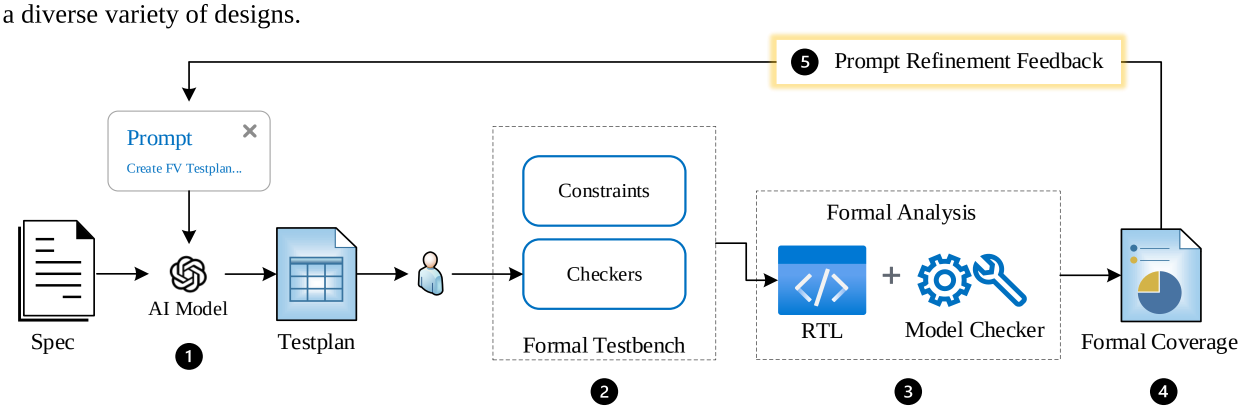

Figure 4: Completion Evaluation & Prompt Refinement Framework

This framework accepts user prompt(s) with a specification document, which outlines the requirements and functionalities that need to be verified. Prompt(s) and specifications are fed into a generative AI model. The AI model processes the specification based on the prompt(s) to generate a formal property verification testplan. This step involves understanding the requirements and translating them into a structured plan for verification. This testplan includes checkers to ensure the design meets the specified requirements.

The testplan is then used to create a formal testbench using human effort. The formal testbench consists of implementation of checkers as suggested by the testplan and constraints identified through false failures of the implemented checkers. The formal testbench is then used in the formal analysis phase, which involves two main components – (1) RTL implementation, this is the design under verification, represented at the register transfer level and (2) Model Checker, this is an EDA tool checks the RTL against the implemented checkers under the constraints defined in the formal testbench. The results of the formal analysis are used to generate a formal coverage report. This report indicates how thoroughly the design has been verified against the specified requirements.

Based on the formal coverage report, the process may enter a prompt refinement loop. This loop involves refining the initial prompt(s) to address any uncovered logic in the DUT and improve the verification coverage. The refined prompt is then fed back into the generative AI model to generate an updated testplan, and the process repeats. This iterative process ensures that we arrive at a high-quality, robust prompt script that can be deployed in formal verification process of wide variety of designs.

B. *Prompt Building Mechanism*

A prompt is a natural language input to a model to tell it what you want it to do. Prompt engineering [24] is the process of designing queries that steer the model’s outputs toward desired outcomes. It involves crafting specific inputs to guide the model in generating the most relevant and accurate responses. Prompt engineering is a crucial aspect of working with large language models (LLMs) like GPT4, LLaMA, and others. It involves creating well-structured and precise prompts, process akin to giving clear and concise instructions to a human to ensure they understand and perform the task correctly.

Process of crafting prompts is essential for several reasons. It improves model performance – clear and specific prompts, we can reduce ambiguity and ensure that the model understands the context better, leading to more accurate and relevant outputs. It reduces hallucinations – carefully designing prompts can minimize the risk of the model generating incorrect or harmful outputs, also known as hallucinations. This is particularly important in applications where accuracy and reliability are paramount, such as healthcare and autonomous systems. It creates better user experience – well-crafted prompts ensure that the model's responses are more aligned with user expectations, leading to a more satisfying and productive interaction.

Once a generative AI model is selected based on context window size, controllable parameter, cost, quality etc., we follow a 3-step process (shown in figure 5) for building an effective prompt script using the framework detailed above.

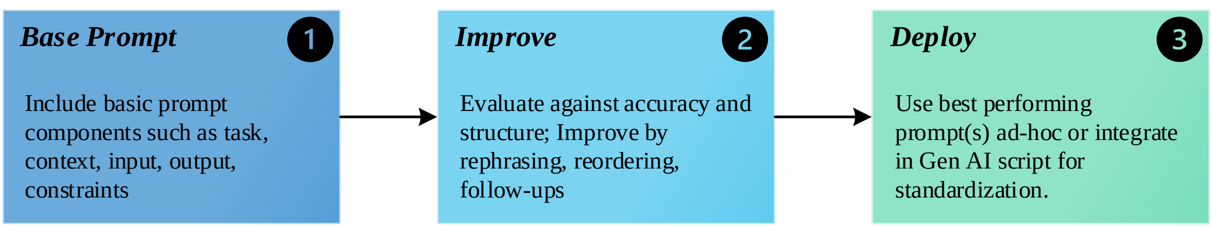

Figure 5: Prompt Building Mechanism

Step-1: Creating an effective prompt for a model involves several key components. Each component plays a crucial role in guiding the model to generate the desired output. Below is a detailed explanation of the basic prompt components, along with examples to illustrate their application.

1. **Context** – Information that helps the model understand the task and generate an appropriate response.
2. **Task** – Guide the model on what to do. It should be specific without much room for interpretation.
3. **Input** – The actual text that the model will read. It can include a question, a statement, a document.
4. **Output** – Description of the expected response including structure, format, or a template.
5. **Constraints** – Any limitations or rules that the model should follow such as tone, style, word count.

Following is an example of building a prompt comprising of basic prompt components for generating a formal property verification testplan.

Table 2: Basic Prompt Components

Prompt Component	Example
Context	*As someone with extensive experience in verifying hardware designs using formal property verification, I would develop a testplan for a design under test.*
Task	*Write an exhaustive list of checkers required to verify each of the design functions.*
Input	*Please go through the specification document and understand the overall functionality of the design “foo” from the perspective of a formal verification engineer.*

Output	*Arrange the list of checkers in a tabular format. Table should have 4 columns. First column should be unique ID for each checker (e.g., CHK1, CHK2, …), second column should be the name of checker, third column should be the detailed description of the checker.*
Constraints	*Checkers should be planned in such a way that they can be implemented using only inputs and outputs of the design.*

Step-2: Evaluating the quality of output generated by generative model essentially means "did the model complete the task I asked for within my constraints, accurately? Or would I like to improve it?". Evaluation criteria could be subjective or objective. For creative tasks, subjective criteria based on common sense, judgement, etc. could suffice. However, for factual tasks, objective criteria based on comparison with ground truth is necessary to evaluate the responses of generative model. Since we are dealing with factual task of spec-to-testplan, we use objective criteria based on formal coverage to evaluate the responses. We will share more details about the evaluation criteria in next sub-sections.

There are many useful advanced prompt design methods that can be applied while iterating over the basic prompt that can help improve the quality of responses from the generative models. Some of such methods includes adding persona [25], providing examples for In-Context Learning (ICL) [26], use Chain-of-Thought (COT) [27] [28] to encourage the model to follow a series of steps in its reasoning which improves its performance on complex tasks and facilitate more context-aware responses, ask the model if it missed anything on previous passes.

Step-3: Select and use the most effective prompts on a case-by-case basis. The goal is to achieve the best possible performance for specific tasks or scenarios. Or incorporate the best performing prompts into a generalized AI script that can be used consistently across various applications.

Following is a sequence of prompts we have perfected for generating high-quality formal verification testplans.

Table 3: Our Suggested Prompt Sequence

#	Prompt
1	*Assume the persona of a formal verification engineer, expert in verifying the functionality of digital circuits using formal property verification to ensure that all the design implementation adheres to design specification.*
2	*Please go through the specification provided as attachment and understand the overall functionality of the design from the perspective of a formal verification engineer. Then, execute the following steps to create a detailed formal verification testplan. Formal verification testplan comprise of list of checkers, constraints, and functional covers. Your description of each checker should be specific instruction on what needs to be implemented by a formal verification engineer. For example, instead saying "FIFO full should be asserted correctly", say "FIFO full should be high when all the entries of FIFO are in use, and FIFO full should be low when all the entries are not in use".*
3	*[Step 1] Summarize the specification in two sections, core functionality of the design and the flow of operation.*
4	*[Step 2] Write an exhaustive list of checkers required to verify each of the design functions. Checkers should be planned in such a way that they can be implemented using only inputs and outputs of the design.*
5	*[Step 3] Arrange the list of checkers, constraints, and functional covers in a tabular format. Table should have 4 columns. First column should be unique ID for each checker (e.g., CHK1, CHK2, …), second column should be the name of checker, third column should be the detailed description of the checker.*
6	*Are these all the checkers? Please take care not to miss or repeat any checker.*
7	*[Step 4] Please suggest a staging plan in which the checkers should be implemented and debugged.*

8	*[Step 5] Arrange the staging plan in a tabular format. It should have 3 columns, Stage, Objectives and Tasks. Tasks should mention the implemented in each stage.*

C. Prompt Evaluation Metric

Testplan creation is a factual tasks, thus requires objective evaluation metric for obtaining accurate results. We leverage formal coverage to evaluate the quality of prompt(s) used in creating formal verification testplans. Formal coverage is a powerful way to gauge the quality of formal testbench used to verify a design. Formal coverage can be categorized as follows:

1. **Observability Coverage** – It measures the exhaustiveness of the checkers implemented. It determines the extent to which the design is being observed and verified for specific properties. Observability coverage helps ensure that the checkers adequately capture the desired properties and behaviors of the design. Observability coverage can be further classified into COI, Formal Core and Mutation coverage [29] based on the precisions, where COI coverage is least precise and Mutation coverage is most precise.
2. **Reachability Coverage** – It determines the extent to which different parts of the design are reachable under a given set of implemented constraints. Reachability coverage helps identify whether all the intended behaviors and states of the design can be reached during the formal verification process.

In our work, we use **Observability Coverage for evaluating exhaustiveness of checkers** suggested in generated testplan. Assuming there are no human errors in implementation of testplan and proofs are converging, observability coverage is a function of completeness of formal verification testplan. As shown in figure 6, each unique and accurate checker increases the observability coverage (increases the exhaustivity score of the testplan). For example, correct checkers CHK1, CHK2, CHK3 raises the observability coverage whereas redundant checker CHK2 and incorrect checker CHK4 has no impact on the observability coverages. Therefore, prompt evaluation based on formal coverage inherently captures the accuracy and exhaustivity of the generated formal verification testplans.

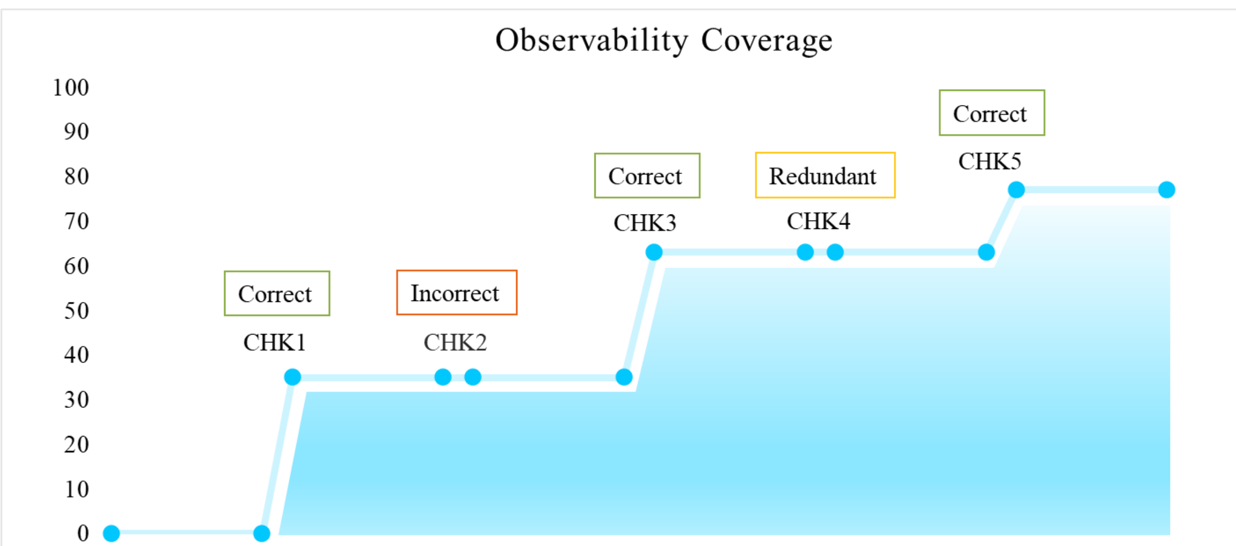

Figure 6: Evaluation of Generated Testplan using Formal Coverage

IV. Experimental Results

We test robustness of our suggested prompt sequence captured in table 3 against three designs of variable design complexity. These designs were identified from open-source projects such as OpenTitan [30] and PULP Platform [31] based on availability of reasonable specification details. All experiments were run on 4 Core, 256G, Intel Xeon CPUs.

A. *Write Packer Design*

The packer module is designed to receive partial writes and pack them into full configurable width writes. The module receives input data along with a bitmask. The bitmask indicates which parts of the data are valid. It accumulates incoming data until the total width of the accumulated data matches a predefined output data width. Then, the accumulated data is output along with a corresponding output bitmask. In case of flush, the module attempts to output any remaining data in its internal storage, even if it does not form a full width write. Complete specification is available at Packer - OpenTitan Documentation.

Design Statistics for Packer (InW=32, OutW=256)

659	Number of Flops
26,954	Number of Gates
450	Number of Nets
13	Number of Ports
699	Lined of Code

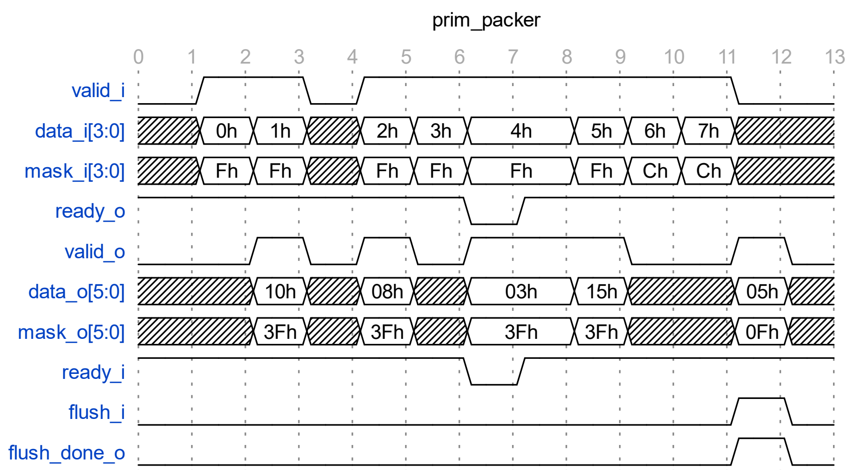

Figure 7: Packer Flow of Operation

Table 4: AI Generated Packer Testplan

ID	Name	Description
CHK1	*Data Packing Checker*	*Verifies that input data (data_i) is correctly accumulated and outputs as data_o once enough data has been gathered to form a full OutW width packet.*
CHK2	*Mask Output Correctness Checker*	*Ensures that the output mask (mask_o) correctly represents the valid bits of data_o, indicating full width writes or partial writes during a flush operation.*
CHK3	*Flush Operation Completeness Checker*	*Confirms that upon assertion of flush_i, all stored data is output, and flush_done_o is asserted once the internal storage is emptied.*
CHK4	*Ready Signal Behavior Checker*	*Validates the ready_o signal behavior, ensuring it correctly indicates when the module can or cannot accept further input data.*
CHK5	*Error Signal Assertion Checker*	*Checks that the error output (err_o) is correctly asserted in response to fault injection attacks on the position counter, assuming EnProtection is enabled.*
CHK6	*Misaligned Writes Handling Checker*	*Verifies that misaligned writes, indicated by partial mask_i inputs, are correctly handled and packed into OutW wide outputs.*

CHK7	*Backpressure Propagation Checker*	*Ensures that backpressure from downstream (indicated by ready_i) correctly influences the ready_o signal to prevent new data acceptance.*

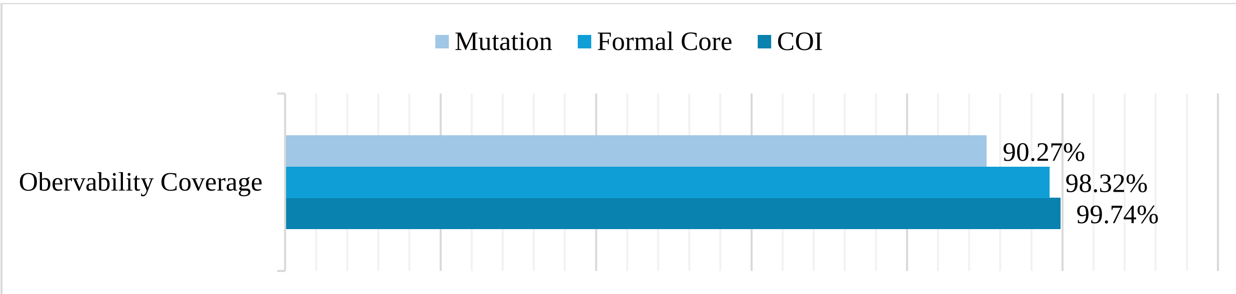

Figure 8: Packer Coverage (excluding deadcode)

B. ID Queue Design

The ID Queue is designed to manage elements with numeric IDs while maintaining FIFO order among elements with the same ID. It provides mechanisms for pushing (inputting) and popping (outputting) elements. The design supports two operational modes – (1) 50% Bandwidth mode where input and output operations cannot be performed simultaneously and (2) Full Bandwidth mode that allows simultaneous input and output operations, enabling a popped cell to be reused in the same clock cycle. The design also supports searching for elements within the queue. The search can be masked, allowing for partial matches based on specified bits. Its specification is available as detailed comment in the RTL file available at common_cells/src/id_queue.sv at master · pulp-platform/common_cells · GitHub.

Design Statistics for ID Queue (ID_WIDTH = 8, CAPACITY=192, DATA_WIDTH=64, FULL_BW = 0)

18,816	Number of Flops
430,975	Number of Gates
176,780	Number of Nets
17	Number of Ports
499	Lined of Code

Table 5: AI Generated ID Queue Testplan

ID	Name	Description
CHK1	*Queue Capacity Checker*	*Ensure the queue does not exceed its maximum capacity (CAPACITY). Formula: assert(queue_size <= CAPACITY);*
CHK2	*FIFO Order Checker*	*Verify that elements with the same ID are dequeued in the order they were enqueued. Formula: assert(oup_id_o == expected_id && oup_data_o == expected_data);*
CHK3	*Push Operation Checker*	*Ensure elements are correctly enqueued when inp_* signals are asserted. Formula: assert(inp_valid_i && !queue_full => element_enqueued);*
CHK4	*Pop Operation Checker*	*Ensure elements are correctly dequeued when oup_pop_i is asserted. Formula: assert(oup_pop_i && queue_size > 0 => element_dequeued);*
CHK5	*Simultaneous Push/Pop Checker (!FULL_BW)*	*Ensure that in !FULL_BW mode, push and pop operations do not occur simultaneously. Formula: assert(!FULL_BW && (inp_valid_i ^ oup_pop_i));*
CHK6	*Simultaneous Push/Pop Checker (FULL_BW)*	*Ensure that in FULL_BW mode, push and pop operations can occur simultaneously. Formula: assert(FULL_BW && inp_valid_i && oup_pop_i);*

CHK7	*Non-Destructive Read Checker*	*Verify that a non-destructive read does not remove the element from the queue. Formula: assert(!oup_pop_i && oup_data_valid_o => element_not_removed);*
CHK8	*Exists Port Checker (Unmasked)*	*Ensure the exists_ port correctly identifies the presence of an element without masking. Formula: assert(exists_req_i && exists_mask_i == '1 && exists_data_i == queue_element => exists_data_valid_o);*
CHK9	*Exists Port Checker (Masked)*	*Ensure the exists_ port correctly identifies the presence of an element with masking. Formula: assert(exists_req_i && (exists_data_i & exists_mask_i) == (queue_element & exists_mask_i) => exists_data_valid_o);*
CHK10	*Output Data Validity Checker*	*Verify that oup_data_o is valid when oup_data_valid_o is asserted. Formula: assert(oup_data_valid_o => oup_data_o == expected_data);*
CHK11	*Input Data Validity Checker*	*Verify that input data is correctly captured. Formula: assert(inp_valid_i => inp_data_i == stored_data);*

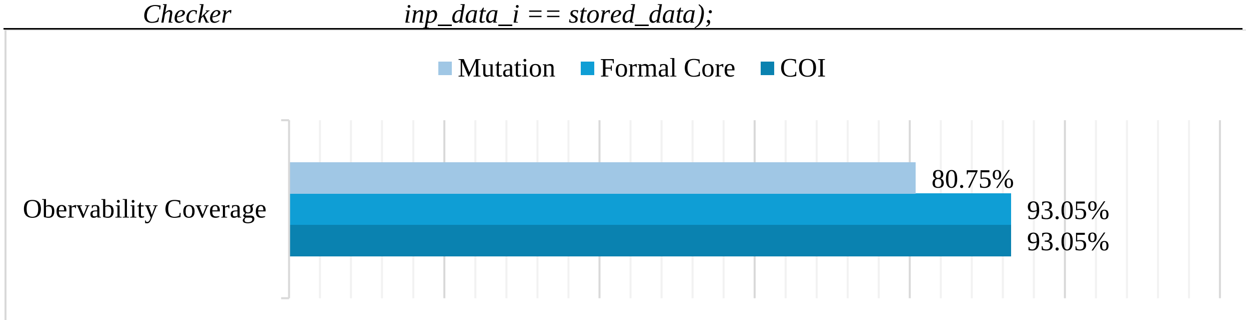

Figure 9: ID Queue Coverage (excluding deadcode)

F. *Xbar Design*

The Xbar design is a fully connected switch that adheres to the AXI4-Lite specification. It enables communication between multiple master and slave modules by providing direct connections between each master port and every slave port. A shared address map determines the routing of transactions from master ports to slave ports. This map includes rules that define address ranges and their corresponding master ports. Multiple rules can overlap, and in such cases, the rule with higher priority prevails. Each slave port has an internal decode error slave module. If a transaction's address does not match any rule, it is routed to this module, which responds with a decode error. Alternatively, a default master port can be designated to handle unmatched addresses, bypassing the decode error module. The crossbar maintains transaction order using a network of FIFOs, ensuring in-order transmission. However, stalls in one module can affect consecutive transactions in flight. Complete specification is available at axi/doc/axi_lite_xbar.md at master · pulp-platform/axi · GitHub.

Design Statistics for Mailbox (NoSlvPorts=16, NoMstPorts=16)

44,096	Number of Flops
4,915,714	Number of Gates
231,213	Number of Nets
67	Number of Ports
4,053	Lined of Code

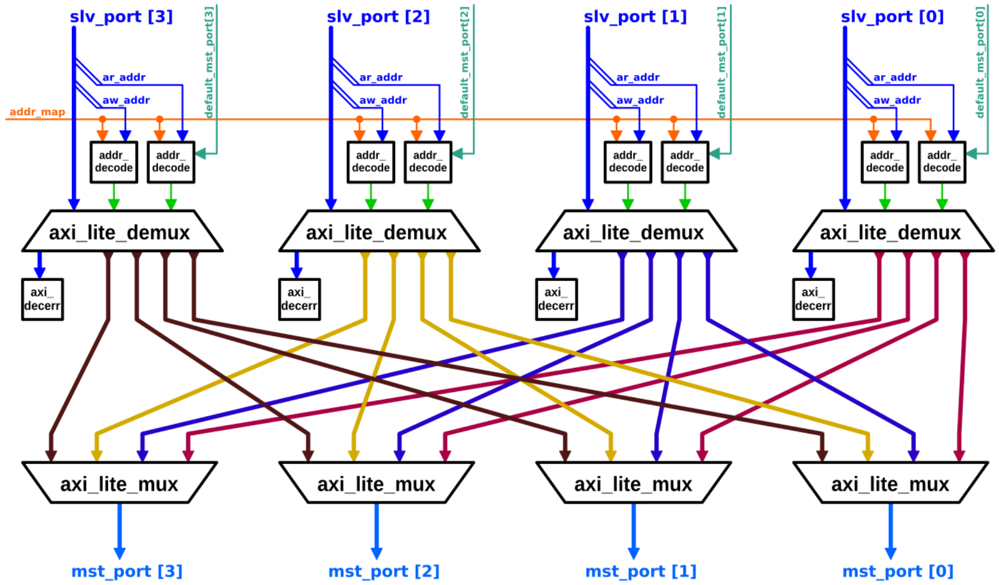

Figure 10: Block Diagram of AXI4-Lite Xbar Design

Table 6: AI Generated Xbar Testplan

ID	Name	Description
CHK1	*Valid Address Routing*	*Ensure that transactions with addresses falling within defined ranges are routed correctly.*
CHK2	*Invalid Address to Decode Error*	*Verify that transactions with addresses not matching any rule are routed to decode error slave.*
CHK3	*Invalid Address to Default Master*	*Confirm that transactions with invalid addresses are routed to the default master port if enabled.*
CHK4	*Master to Slave Connectivity*	*Ensure that each master port can send transactions to all slave ports.*
CHK5	*Slave to Master Response*	*Verify that responses from slave ports are correctly routed back to the originating master port.*
CHK6	*Default Master Enable*	*Ensure that when the default master port is enabled, transactions with unmatched addresses are routed to it.*
CHK7	*Default Master Disable*	*Confirm that when the default master port is disabled, transactions with unmatched addresses are routed to decode error slave.*
CHK8	*Decode Error Response Data*	*Verify that the decode error slave provides the expected data pattern for read transactions with invalid addresses.*
CHK9	*Decode Error Write Handling*	*Ensure that the decode error slave responds with the correct error codes for write transactions with invalid addresses.*
CHK10	*Initial Configuration*	*Confirm that the initial configuration of the crossbar matches the provided axi_pkg::xbar_cfg_t settings.*
CHK11	*Runtime Configuration Changes*	*Verify that any runtime changes to the crossbar's configuration are correctly implemented and take effect as expected.*
CHK12	*Valid Transaction Format*	*Ensure that all transactions confirm to the AXI4-Lite protocol specifications.*

CHK13	*Transaction Completion*	*Verify that all transactions initiated by masters reach a completion state without hanging.*
CHK14	*Address Range Access*	*Ensure that all address ranges specified in the address map are accessible and correctly routed to the appropriate slave ports.*
CHK15	*Unreachable Address Detection*	*Verify that there are no address ranges in the configuration that are unreachable by any master port.*
CHK16	*Transaction Latency*	*Ensure that the latency of transactions from masters to slaves and back is within specified limits.*
CHK17	*Throughput Consistency*	*Verify that the throughput of transactions remains consistent even under different load conditions.*

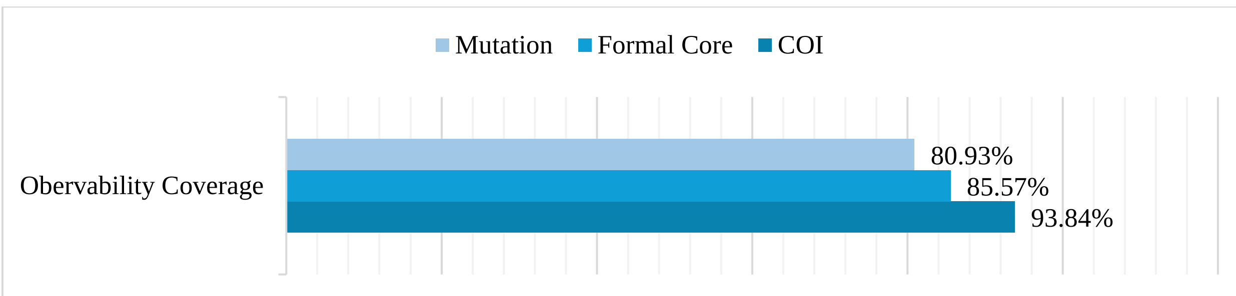

Figure 11: Xbar Coverage (excluding deadcode)

V. Discussion

A. Efficacy Analysis

Our tests on small, medium and high design complexity blocks suggests our robust prompt sequence is consistent in generating good quality formal verification testplan which once implemented can **cover more than 80% of the design functionality**, reducing the human effort in making such testplans from scratch. On even larger designs, our methodology can bring-down several weeks of efforts to a couple of hours saving a great deal of human effort. Generated testplans are structured in a tabular format which is easy to export of spreadsheet-based tracking or other tracking apps offered by EDA vendors. Each row of the generated testplan assigns a serial number, a name as well as a reasonable description for the verification engineer to interpret the checker that needs to be implemented.

B. Common Limitation

We have observed that some of the checkers suggested in the generated testplans are redundant. These redundant checkers seem to cover a subset of logic which has already been checker by other suggested checkers. This limitation can increase the overall checker development effort. However, careful review of the generated testplan can mitigate the effect of redundancy. In some cases, we also observed hints of simulation-based verification testplan especially in case of suggestions to verify initial hardware configuration of the DUT. These suggestions are valid but not applicable to formal verification and can be easily ignored by formal verification engineers.

C. Area of Further Research

One of the major are research that can help advance the efficacy of AI generated formal verification testplan is to integrate RAG [32] to the proposed approach. Retrieval-Augmented Generation (RAG) aims to optimize the output of generative models by referencing knowledge base outside of its training data sources before generating a response. RAG extends the already powerful capabilities of generative models to specific domains or an organization's internal knowledge base, all without the need to retrain the model.

V. Conclusion

In this paper, we introduce a mechanism to generate high-quality formal verification testplan using generative AI models using the principles of prompt engineering. We offer an innovative framework to develop quality prompt sequences that can be deployed to real-world formal verification projects. Our proposed framework is objective in evaluation, flexible to user's needs, scalable for different size and complexity of designs, and tested to produce 80%+ accurate and exhaustive formal verification testplans.

REFERENCES

[1] Pete Hardee, "Choosing the Right Superlinting Technology for Early RTL Code Signoff".

[2] Atsunori Machinda, "Fujitsu: Efficient Static Verification Sign-off for High-value Failure Mode" Design Automation Conference (DAC).

[3] Rohan Sawant, Shylaja Sen, "VC SpyGlass Lint: Early, Automated Detection of Design Issues".

[4] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "Autosva: Democratizing formal verification of rtl module interactions," in 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021, pp. 535–540.

[5] W. Fang, G. Hu, and H. Zhang, "r-map: Relating implementation and specification in hardware refinement checking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023.

[6] S. Germiniani and G. Pravadelli, "Harm: a hint-based assertion miner," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 11, pp. 4277–4288, 2022.

[7] A. Danese, N. D. Riva, and G. Pravadelli, "A-team: Automatic templatebased assertion miner," in Proceedings of the 54th Annual Design Automation Conference 2017, 2017, pp. 1–6.

[8] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). IEEE, 2010, pp. 626–629.

[9] C. B. Harris and I. G. Harris, "Glast: Learning formal grammars to translate natural language specifications into hardware assertions," in 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016, pp. 966–971.

[10] R. Krishnamurthy and M. S. Hsiao, "Controlled natural language framework for generating assertions from hardware specifications," in 2019 IEEE 13th International Conference on Semantic Computing (ICSC). IEEE, 2019, pp. 367–370.

[11] 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). IEEE, 2019, pp. 598–601.

[12] S. J. Frederiksen, J. Aromando, and M. S. Hsiao, "Automated assertion generation from natural language specifications," in 2020 IEEE International Test Conference (ITC). IEEE, 2020, pp. 1–5.

[13] O. Keszocze and I. G. Harris, "Chatbot-based assertion generation from natural language specifications," in 2019 Forum for Specification and Design Languages (FDL). IEEE, 2019, pp. 1–6.

[14] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat et al., "Gpt-4 technical report," arXiv preprint arXiv:2303.08774, 2023.

[15] Erik Seligman, Tom Schubert, M. V. Achutha Kiran Kumar, "Formal Verification – An Essential Toolkit for VLSI Design".

[16] IEEE Std 1800™-2017, IEEE Standard of SystemVerilog – Unified Hardware Design, Specification, and Verification Language.

[17] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, Siddharth Garg, "VeriGen: A Large Language Model for Verilog Code Generation".

[18] M. Li, W. Fang, Q. Zhang, and Z. Xie, "Specllm: Exploring generation and review of vlsi design specification with large language model," arXiv preprint arXiv:2401.13266, 2024.

[19] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, Zhiyao Xie, "AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs"

[20] Jansen, F. (2006). How bulleted lists and enumerations in formatted paragraphs affect recall and evaluation of functional ext. Information Design Journal. 21, 146-162.

[21] Mirko Thanlmann, A. S. (2019). How does chunking help working memory? Journal of Experimental Psychology: Learning, Memory, and Cognition, 37-55.

[22] Derong Xu and Wei Chen and Wenjun Peng and Chao Zhang and Tong Xu and Xiangyu Zhao and Xian Wu and Yefeng Zheng and Yang Wang and Enhong Chen, "Large Language Models for Generative Information Extraction: A Survey".

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

[24] Xavier Amatriain, "Prompt Design and Engineering: Introduction and Advanced Methods".

[25] Murray Shanahan and Kyle McDonell and Laria Reynolds, "Role-Play with Large Language Models".

[26] Katerina Margatina and Timo Schick and Nikolaos Aletras and Jane Dwivedi-Yu, "Active Learning Principles for In-Context Learning with Large Language Models".

[27] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, Denny Zhou, "Chain of Thought Prompting Elicits Reasoning in Large Language Models".

[28] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Denny Zhou, "Self-Consistency Improves Chain of Thought Reasoning in Language Models.".

[29] Pallab Dasgupta, "A roadmap for formal property verification".

[30] GitHub - lowRISC/opentitan: OpenTitan: Open source silicon root of trust

[31] pulp-platform · GitHub

[32] Patrick Lewis and Ethan Perez and Aleksandra Piktus and Fabio Petroni and Vladimir Karpukhin and Naman Goyal and Heinrich Küttler and Mike Lewis and Wen-tau Yih and Tim Rocktäschel and Sebastian Riedel and Douwe Kiela, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”.