LLM-based Functional Coverage Generation and Auto-Evaluation Framework

Jan Labuda
Masaryk University
Brno, Czech Republic
jan.labuda@mail.muni.cz

Marcela Zachariasova
Brno University of Technology
Brno, Czech Republic
zachariasova@fit.vut.cz

Zdenek Matej
Masaryk University
Brno, Czech Republic
matej.zdenek@mail.muni.cz

***Abstract*—To ensure that a Design Under Verification (DUV) is thoroughly examined during the simulation-based verification process, one of the metrics verification engineers may rely on is functional coverage. This metric is manually implemented in the testbench and tracks which functionalities have been exercised in DUV during testing. Large language models (LLMs) have recently shown potential in automating code generation across various domains. This paper investigates their capabilities to transform verification requirements written in natural language into syntactically and semantically correct functional coverage code, as this domain remains underexplored. To accomplish this, an automated evaluation framework was developed to assess several open-weight LLMs' performance on this task for a reference computational design. The goal was to have a highly controllable evaluation environment for these initial experiments. The results reveal promising capabilities of LLMs in this context, while also identifying challenges and limitations where their performance fell short. The authors provide insights into the underlying reasons for these difficulties, contributing to the understanding of LLMs' potential and limitations in verification tasks.**

***Index Terms*—Universal Verification Methodology (UVM), Functional verification, Coverage, LLM, CoCoTB, Benchmark**

I. Introduction

Functional coverage metric plays an important role in maintaining the integrity of simulation-based verification testbenches. It determines if the implemented tests have effectively exercised the intended functionality. Unlike code coverage metrics, which focus on whether the tests have exercised various structures in the source code, functional coverage ensures that the tests cover all functional corner cases and intriguing combinations of signal values on interfaces [1]–[3]. For instance, if the implementation of an arithmetic logic unit is missing an operation described in the specification, the code coverage may be fulfilled. However, functional coverage could reveal the missing functionality, thereby preventing potential issues from going unnoticed.

In practice, functional coverage is implemented by verification engineers based on functional implications extracted from the design specification, which are usually expressed by functional verification requirements in the verification plan. Verification requirements may be expressed in natural language, or they can follow a structured, machine-readable format [4], [5], be template-based (XML, YAML, JSON, etc.) [6], or even formally expressed [7]. A single project can easily contain hundreds of verification requirements. Figure 1 depicts the standard verification workflow. In the first step, a verification plan with functional requirements is derived from the design specification. Following that, a verification team implements verification tests used for stimulating a behavior in DUV, which reflects the verification requirements, a testbench (such as a UVM-based testbench in SystemVerilog, or a Python-based Cocotb testbench) which executes the tests and evaluates their results, and functional coverage to provide feedback about functionality which was covered by the tests.

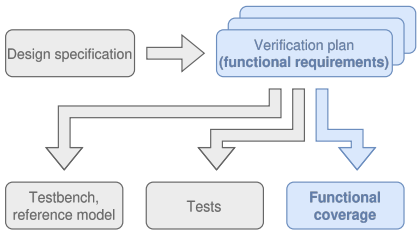

Fig. 1: The conventional workflow in verification. Highlighted parts in blue are targeted in this paper.

While functional coverage is a useful metric that provides valuable feedback to verification engineers, it often does not receive the same level of importance as the reference model with predictors, various checkers, or tests. This may explain why some projects skip its implementation or assign it a low priority. Therefore, incorporating functional coverage in the testbench would be beneficial, particularly if it could be generated automatically to alleviate the burden of manual implementation.

This paper's primary objective is to explore LLMs' capabilities in generating functional coverage code based on design specifications or functional requirements articulated in natural language, as this would rapidly simplify the verification processes. We chose to construct the input dataset for LLMs around verification requirements in a specific format, rather than directly relying on design specifications, even though they are an essential source of information. This decision was influenced by the varying formats, inconsistency of provided information, and the overall quality of design specifications currently freely available. More importantly, ongoing research focusing on extracting valuable information directly from design specifications is a research topic of its own, for example, the work presented in [8], which, if successful, could be incorporated into our work. Therefore, it seemed prudent to narrow the scope of our research to provide more reliable outputs; hence, we opted to bypass the specification and to work with verification requirements as an input for LLMs.

The main contributions of this paper, including the key research steps, are as follows.

- Assesses LLMs' capabilities in capturing the information articulated in verification requirements to generate functional coverage.
- Evaluates the ability of LLMs to produce sufficiently complex functional coverage code that offers valuable feedback during the verification runs.
- Outlines a detailed auto-evaluation framework for assessing the solutions generated by the LLMs.

II. Related Work

Previous studies have suggested generating functional coverage based on features extracted from the DUV implementation [9], [10]. However, relying on the DUV implementation instead of the specification poses risks. The generated code is closely tied to the actual functionality of the DUV, which means it may overlook missing implementations of functionalities described only in the specification.

Other approaches use deterministic generation algorithms to transform structured verification requirements into functional coverage code. For instance, the method presented in [5] begins with table-based, ASCII-encoded state tables that outline the protocol's legal and illegal state transitions across different agents. These state table items are then translated into functional covergroups, coverpoints, and crosses. The generated coverage is checked using a formal tool to ensure reachability of all table-specified states and transitions; unreachable cases are flagged for review. Similar work is detailed in [11]. Also, in this case, the tables do not represent raw functional or natural-language requirements. The authors assume that the verification engineer has already encoded the intent into a structured, tabular format, effectively creating a semi-formal coverage plan.

Work presented in [4] uses a high-level executable specification model, which serves as a structured, programmatic representation of functional behaviors, states, sequences, and transactions, from which functional coverage code can be deterministically generated.

In summary, the benefits of the above-mentioned methods are evident: the coverage is deterministically generated, and if the requirements are accurately written, it is possible to obtain working functional code. However, each method requires a distinct format of verification requirements, which are typically highly structured and developed by engineers from natural language specifications.

In today's era of LLMs and AI-assisted programming, it is timely to revisit the topic as LLMs have demonstrated their ability in generating code from tasks articulated in natural language across different domains [12]. Authors of the paper [13] discuss the ability of LLMs to generate SystemVerilog UVM testbench and input stimuli based on the provided DUV interface, DUV description, and testbench structural organization. They use various coverage metrics to evaluate the generated solutions. Their evaluation reveals that open-weight LLMs struggle with complex design understanding, often achieving only basic metrics like line coverage. In contrast, this paper focuses on selectively generating testbench components that are simpler and less prone to misunderstanding. The generation of functional coverage structures was identified as an ideal starting point. Moreover, to our knowledge, no existing work specifically uses LLMs to convert natural language verification requirements into functional coverage structures; thus, this paper presents a novel approach.

III. Background

This section addresses functional coverage implementation, coverage measurement during simulation, and generation of functional coverage code using LLMs.

A. Functional coverage

Figure 2 illustrates the distinction between measuring functional coverage and code coverage. Code coverage is automatically tracked in the RTL simulator, where both the testbench and the DUV operate. In contrast, functional coverage must be explicitly implemented within the UVM testbench. This is typically accomplished through a UVM coverage monitor or UVM collector component that receives transactions via analysis connections from UVM monitors.

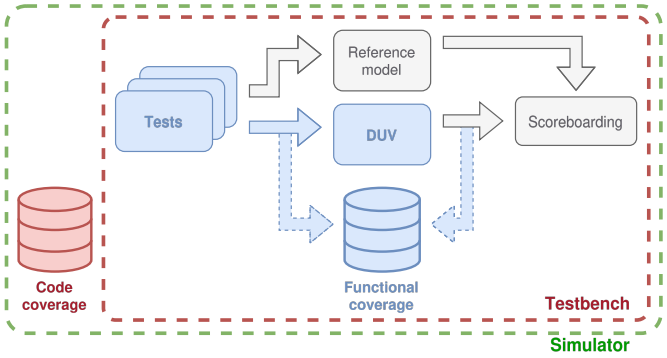

Fig. 2: A simplified UVM testbench structure including functional (blue) and code (red) coverage targets.

The coverage monitor samples the values of specific transactions' signals or variables at appropriate moments during simulation. The central element of this implementation is referred to as a covergroup, which acts as a container for defining coverpoints and cross coverpoints. A coverpoint is linked to a single variable from a transaction and employs structures known as bins to record the occurrence of specific values or ranges of values throughout the simulation. Besides the bins capturing individual values, the coverpoint can also register a transition between values (e.g., a change from zero to one on a reset signal in consecutive clock cycles). Cross coverpoints, on the other hand, capture combinations of values from two or more coverpoints, enabling the evaluation of interactions between different variables reflecting corner cases in DUV.

In our work, LLMs aim to generate a suitable representation of covergroups with reasonable coverpoints and cross coverpoints to adequately represent the verification requirements from the verification plan, which will be provided as their input.

B. Large Language Models

Widely researched in recent years, LLMs are a type of language model characterized by hundreds of millions to hundreds of billions of parameters. Trained on vast amounts of text data, they are capable of generating text across a wide range of domains, including solving programming tasks [12].

Even though they are a promising candidate for generating functional coverage code, the two main concerns, namely data security in the case of using LLM behind a third-party API and noticeable knowledge bias across various domains, must be taken into account. Considering the criticality of data security in chip design companies, they often prohibit access to an LLM of a third party. The latter concern is related to the uneven spread of training textual data across various domains. For example, according to the *RepositoryStats*[1], only 0.05% of the captured public *GitHub* repositories have *SystemVerilog* as their primary language. Both of these concerns might be addressed using an open-weight finetuned LLM that can be run locally on consumer-grade hardware.

An example of finetuned models is *RTLCoder*, consisting of just 7 billion parameters, which managed to outperform even much larger GPT-4 model in VerilogEval benchmark [14]. In case of chip design verification, there is an automated framework *Assert-O* designed to generate security properties from documentation, finetuned on data from *OpenTitan* project [15].

[1]https://repositorystats.com/language/systemverilog

Considering that the aim of this paper is to evaluate LLMs with respect to the generation of functional coverage from verification requirements, and given the tendency of LLMs to more accurately generate code in widely represented languages on the internet, such as *Python*, compared to less prevalent languages like *SystemVerilog*, we used *Python* as the target language for functional coverage generation instead of the industry-standard *SystemVerilog*. This decision was made to minimize the influence of familiarity with the programming language, which is not relevant to the ability to extract information from verification requirements and to produce corresponding functional coverage code. In the future, LLMs may be fine-tuned to address current limitations in generating functional coverage in *SystemVerilog*. For the purpose of this study, however, *Python* provides a more suitable alternative.

IV. Experimental setup

To ensure multiple LLMs are evaluated in an objective manner, an automatic evaluation framework was created. For every DUV, the inputs into the framework are principally the same: verification requirements written in natural language, coupled with a desired functional coverage code prepared by a verification expert, which is used as the reference implementation, and the source code of the DUV written in SystemVerilog. All necessary steps from the design of the dataset used for evaluation, the construction of a universal testbench, to the automatic evaluation metrics are presented in this section. The framework and dataset, containing the DUV, design specification, and verification requirements, are available on the author's GitHub [16].

A. Dataset

Our current dataset includes a single computational design: an arithmetic-logic unit (ALU). This design was chosen for initial and controllable experiments with LLMs because it allows specifying a variety of verification requirements. Despite being a relatively simple design, we can already challenge LLMs to generate all possible functional coverage structures for it: simple coverpoints with various bins, transition coverpoints, and even cross coverpoints.

The ALU module performs arithmetic and logic operations based on a 4-bit input operation code (OP), see Table I. It processes two 32-bit input operands: REG_A (register) and a second operand selected using the MOVI selector, which chooses between REG_B (register), MEM (memory), and IMM (immediate). The result of a multiplication is provided over two clock cycles: the lower 32 bits are sent first, followed by the upper 32 bits in the next cycle. Output signals include the result (DATA), a result validity flag (VLD), and a ready flag (RDY) indicating when the ALU can accept a new operation. Except for these inputs and outputs, CLK (clock signal), RST (reset signal), and ACT (ALU activation signal) are present as inputs to the ALU.

Based on the ALU specification, an expert in verification defined 16 appropriate verification requirements. Each requirement is a short statement written in natural language, consisting of a single sentence or a few sentences. In addition, the expert also included a sample of code implied by the requirements that was used later in the evaluation phase. Each requirement is categorized according to its target into one of these categories: bin (check specific values), sequence (targeting a transition between values), or cross (checking occurrences of specific values simultaneously). Representative samples for each category are shown in Figure 3.

OP Code (Binary)	Description
0000	Addition
0001	Subtraction
0010	Multiplication
0011	Logical right shift of operand_B
0100	Logical left shift of operand_B
0101	Bit right rotation of operand_B
0110	Bit left rotation of operand_B
0111	Bitwise NOT of operand_B
1000	Bitwise AND
1001	Bitwise OR
1010	Bitwise XOR
1011	Bitwise NAND
1100	Bitwise NOR
1101	Bitwise XNOR
1110	Increment operand_B by 1
1111	Decrement operand_B by 1

TABLE I: Opcode table for ALU operations

Verification requirement (bin)

Cover different ranges of values for the first operand represented by the signal REG_A. Specifically, create a separate bin for: value 0, for a single value from range [1:5], MAX_VALUE, and a single value from range [MAX_VALUE - 5:MAX_VALUE].

Verification requirements (sequence)

Cover transitions 0 -> 1 and 1 -> 0 on the RST signal, each transition should occure at least 5 times.

Cover transitions 0 -> 1 -> 0 and 0 -> 1 -> 1 -> 0 on the VLD signal.

Verification requirements (cross)

Cross cover all possible ACT signal values with all possible RST signal values.

Cross cover a small value from the range [0:1000] of registers REG_A and REG_B.

Fig. 3: Examples of the verification requirements.

B. Functional coverage generation

The pipeline used for the generation of functional coverage code from the dataset, featuring the prompt template with an example of verification requirement, is shown in Figure 4. In an attempt to maximize the performance of LLMs, multiple prompt engineering techniques were used in the process of code generation, including a verification role assignment and the few-shot prompting method [17], where LLMs were provided with two examples expalining the generation of code for bins with cross coverpoint and bins for transitions between specific signal values.

To provide an LLM with a chance to fix its code when the generated code contains syntax or typing errors, the code is run through static linters and type analyzers, and the error message is sent back to the LLM. The process of checking the generated code and possibly providing feedback to the LLM is repeated at most three times, taking the first syntactically valid code as the outcome.

After the syntactically valid code was generated, additional checks were performed concerning the computational feasibility and correctness of the generated code. The former check simply ensures that the number of bins contained in the final coverage code is reasonable. In our case, the number of bins is bounded to be at most 1000. The latter check related to the correctness validates that the port names in the generated code match exactly the names of the DUV ports. In addition, all values are checked to be within the range of possible values based on the port bit-width to which they are assigned.

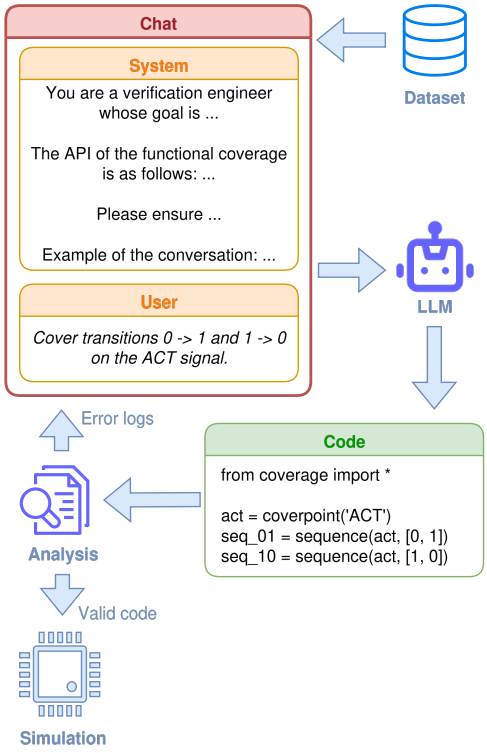

Fig. 4: Generation of the functional coverage code from the dataset.

C. Cocotb testbench

The core component of the auto-evaluation framework utilized in our experiments is represented by a universal Cocotb testbench depicted in Figure 5. Its universality is represented by the ability to evaluate any design without making any changes to its code. The testbench workflow can be divided into three phases: initialization, simulation, and coverage capture.

The initialization phase includes the following steps:

1) Automatically identify all ports of the DUV.
2) Generate a clock signal on an autodetected clock port.
3) Generate an active pulse on an auto-detected reset port.
4) Load the functional coverage code generated by LLMs.

Following initialization, the simulation phase generates uniformly distributed random stimuli to the input ports, ignoring the reset and clock ports. An impulse on the reset port is generated with a probability of 1% at each clock cycle. After the stimuli are assigned, a DUV simulation is run for one clock cycle, and the whole process is repeated for a predetermined time. To make the auto-evaluation framework, including the testbench, more accessible and executable, an open-source simulator *Verilator* [18] was utilized. However, the simulator can be easily changed to a different one supported by *cocotb*[2].

The final phase focuses on capturing both functional and code coverage results during simulation. Currently, functional coverage is sampled at each clock cycle, provided that the reset signal is not active. More sophisticated sampling is planned for future research, with the objective of identifying optimal sampling conditions based on an analysis of input verification requirements.

[2]https://docs.cocotb.org/en/stable/simulator_support.html

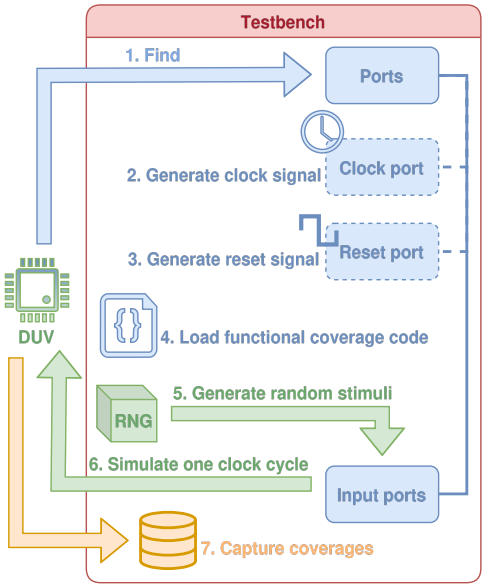

Fig. 5: Cocotb testbench capturing both functional and code coverage. Blue color represents the initialization part, green color the simulation part, and the orange color the coverage capturing part.

D. Automated evaluation

To ensure LLMs generate code that is not only syntactically correct but also covers a reasonable functionality of the DUV, several aspects of the generated output were evaluated:

- **Syntax correctness:** Verifying that the code is free of any syntactic errors and can be executed.
- **Computationally feasible:** Ensuring that the generated coverage code does not try to exhaustively cover the whole state space defined by input variables but rather selects reasonable representative values or ranges.
- **Accuracy**: Assess the extent to which the generated code aligns with the intended functional coverage as defined by the verification requirements.
- **Complexity**: Evaluating how comprehensively the code exercises the DUV, favoring solutions that achieve broader functionality coverage while being computationally feasible.

The first two aspects must be evaluated before a test run. The **syntax correctness** of generated code is checked right after the code is generated, with a possibility of letting LLM fix the issues contained in the code caught by static analyzers. The resulting metric for the syntax correctness is the number of attempts used to generate syntactically valid code, where the maximum number of attempts was set to 3, excluding the initial generation without feedback from static analyzers. **Computational feasibility** is ensured by verifying that the number of specific values, ranges, sequences, or cross products does not exceed a threshold (set to 1000).

The **accuracy** is captured by a Boolean flag related to the similarity of the generated code to the reference. This flag is set to true only if all the required functional coverage provided by the expert contained in the dataset was also present in the generated code.

The last important aspect of evaluation is the **complexity**. The generated coverage should be complex enough and should not miss any functionality. Our solution is based on observations presented in many verification books such as [1]–[3]. It is always suspicious when functional coverage is fully achieved sooner than code coverage,

or if it has higher values in the area of their convergence, see Figure 6. This implies by principle that the functionality expressed in the functional coverage code is not exhaustive and is missing some features. Therefore, our auto-evaluation framework compares achieved functional and code coverage whenever statement coverage is sampled during simulation, targeting a convergence region set to 90% statement coverage.

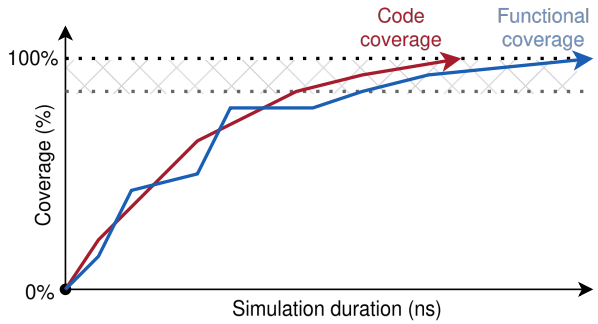

Fig. 6: Illustration of the acquired code (red) and functional (blue) coverage during the simulation. The convergence region is marked with the crossed area.

V. Results

The three most popular open-weight LLM families available at *Ollama*[3] were chosen for evaluation: *Deepseek-r1*, *Qwen3*, and *Gemma3*. Multiple LLMs that differ in the number of parameters were chosen from each family. The naming notation for each model consists of the family name and the number of parameters separated by a colon. For example, *Deepseek-r1:14b* refers to the LLM from *Deepseek-r1* family containing 14 billion parameters. To ensure that the models are viable to run locally on consumer-grade hardware, the size of every used model was kept below 10GB of VRAM. In total, 80 independent pipeline runs were evaluated, comprising 5 attempts for each of the 16 verification requirements.

The total time required to generate 80 functional coverage code snippets is reported in Table II. All models were executed locally on an *NVIDIA A100 80GB PCIe* GPU. The noticeably longer generation times observed for the *Deepseek-r1* and *Qwen3* model families, compared to *Gemma3*, can be attributed to their *thinking* behaviour, where such models produce substantially more intermediate text before making the final answer. During our experiments, there were problems with thinking models, where they had a tendency to get stuck in the thinking phase, not being able to generate the results in a reasonable time. Therefore, the following models had to be omitted: *Deepseek-r1:1.5b*, *Deepseek-r1:8b*, and *Qwen3:1.7b*.

The first two evaluated aspects were the syntactic correctness and viability of the generated functional coverage, where viability refers to computational feasibility and the use of valid ports. The results corresponding to these criteria are summarized in Table III, where values were aggregated to 5 instances each representing an individual generation run using our dataset. Among the evaluated models, both *Gemma3:12b* and *Qwen3:14b* achieved the highest success rate. Notably, *Gemma3:12b* reached comparable performance approximately five times faster.

Figure 7 presents the number of attempts required to generate syntactically correct and viable functional coverage. The top-performing

[3]https://ollama.com

TABLE II: Total time required to generate functional coverage code

LLM	Total Time (MM:SS)	Speed (req./s)
Deepseek-r1:7b	39:32.9	0.03
Deepseek-r1:14b	27:04.4	0.05
Gemma3:1b	5:41.9	0.23
Gemma3:4b	2:23.9	0.55
Gemma3:12b	3:45.7	0.35
Qwen3:1.7b	27:41.2	0.05
Qwen3:4b	11:25.6	0.12
Qwen3:8b	15:16.5	0.09
Qwen3:14b	19:50.5	0.07

TABLE III: Number of syntactically correct and viable code per generation instance (higher is better)

LLM	Syntactically correct			Viable		
	Best	Worst	Mean	Best	Worst	Mean
Deepseek-r1:7b	12	6	10.6	11	5	8.4
Deepseek-r1:14b	**16**	14	15.4	14	13	13.6
Gemma3:1b	11	7	9.2	9	6	7.6
Gemma3:4b	**16**	15	15.4	11	10	10.4
Gemma3:12b	15	**15**	15.0	**15**	**14**	*14.4*
Qwen3:1.7b	12	10	11.4	10	7	8.6
Qwen3:4b	**16**	**16**	**16.0**	14	13	13.6
Qwen3:8b	**16**	**16**	**16.0**	14	**14**	14.0
Qwen3:14b	**16**	**16**	**16.0**	**15**	**14**	**14.4**

models in this regard are *Qwen3:14b* and *Gemma3:12b*, successfully producing 71 and 70 valid snippets, respectively, on the first attempt. A general trend is also observed: models with a larger number of parameters tend to generate valid outputs with fewer attempts.

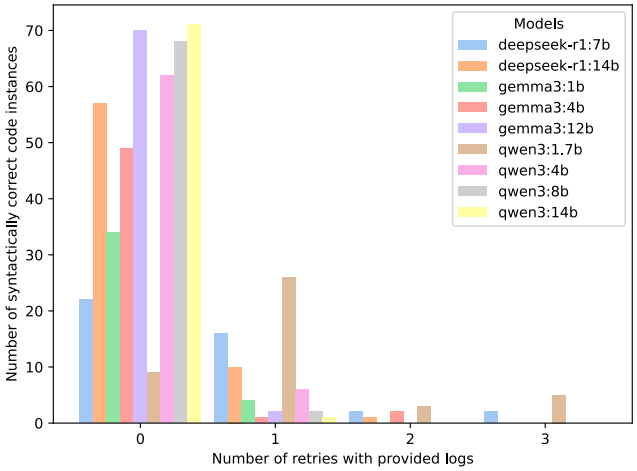

Fig. 7: Plot showing the number of required tries to produce syntactically correct code.

In terms of complexity, all generated functional coverage instances exhibited greater granularity compared to the statement coverage. This can be attributed to the low probability of triggering specific values within wide registers when using randomly generated stimuli from a uniform distribution. Achieving such coverage typically requires targeted or constrained stimulus generation.

The last aspect was accuracy, computed by comparing the generated code to the code provided by the verification expert. Figure 8 summarizes the accuracy of each model based on the generated instances. The most accurate model was *Qwen3:14b*, which in the

best case managed to obtain 93.75% accuracy while its average accuracy was 86.25%. In addition, even a smaller model from the same family, *Qwen3:4b*, managed to outperform the larger model *Deepseek-r1:14b* in accuracy.

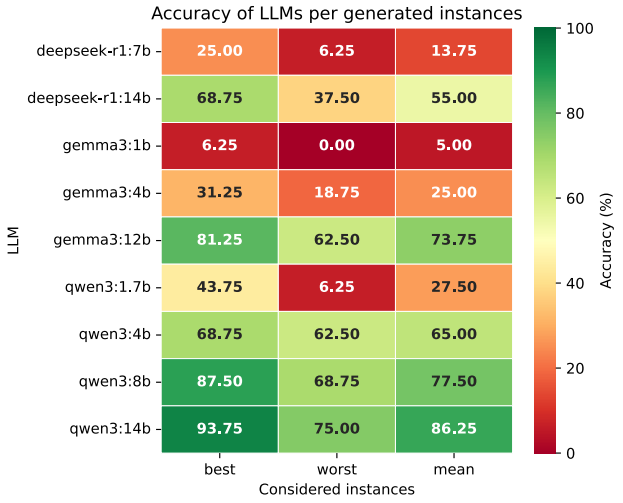

Fig. 8: Accuracy of the LLMs generating functional coverage code among all instances.

Considering the verification target of each verification requirement, the most problematic ones were targeting the cross coverage code constructs, as shown in Figure 9. Even though the *Qwen3:14b* model was shown to be the most accurate one in general, *Gemma3:12b* was more accurate in the generation of code targeting the cross coverage constructs, where its accuracy was better by 3.33%.

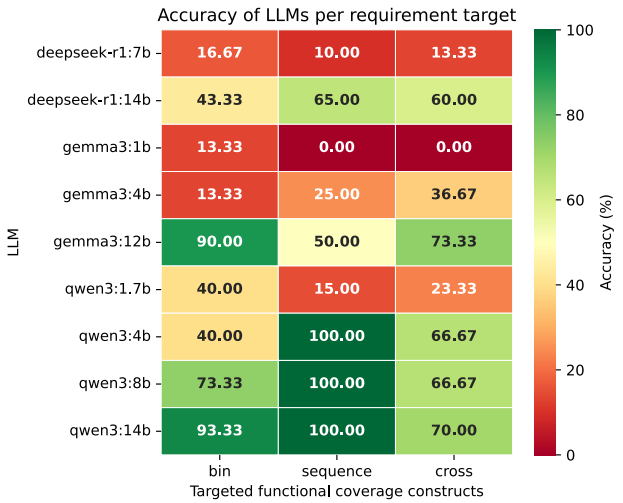

Fig. 9: Accuracy of the LLMs generating functional coverage code based on a requirement target.

The number of generated code snippets that contained the functionality specified by the verification expert out of a total of 5 attempts is depicted in Figure 10. Based on the Figure 10, the most problematic requirements, with IDs 14 and 16, were:

- Cross cover a small value from the range [0:1000] of registers REG_A and REG_B.
- Cross cover that the register DATA contained a value from the range [0:1000] for the first four operations from signal OP.

A common problem that occurred during the code generation for these requirements was that the final code used all values within the range [0:1000] individually, instead of using a single value from that range.

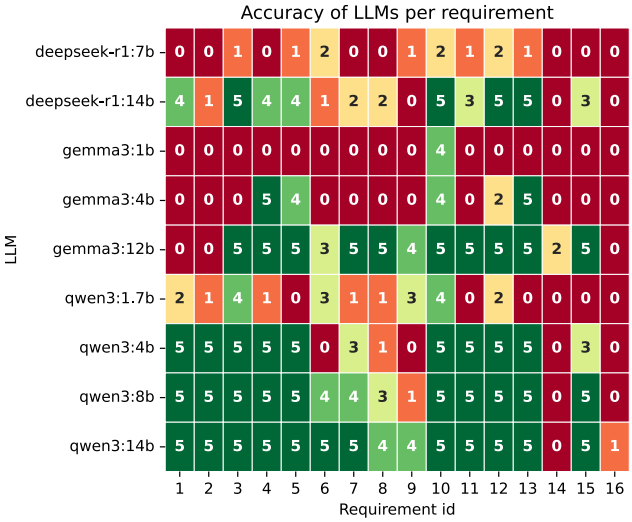

Fig. 10: Number of generated code snippets matching the code provided by the verification expert.

Considering the overall accuracy of each model, *Qwen3:14b* emerges as the most suitable choice for functional coverage generation. Although this model is significantly slower compared to models within the *Qwen3* family, which exhibit only slightly lower accuracy, the generation of functional coverage is typically a one-time process that can be performed overnight. As such, execution speed becomes a secondary concern compared to output quality, making *Qwen3:14b* the better choice.

VI. Conclusion

This paper builds a foundation for the auto-evaluation of functional coverage code generated by available open-weight LLMs from requirements defined in natural language. The main contribution is the framework, which is freely provided for the verification community. The initial results show that some selected open-weight LLMs are capable of extracting most of the information from the verification requirements and generating syntactically correct code in most cases. The 14 billion parameter LLM from the Qwen3 family managed to generate at least 12 out of 16 verification requirements that matched the code provided by the verification expert in each of its attempts, where in its best attempt, it differed only in a single requirement.

The future work will focus on building a much larger dataset containing multiple designs, their specification, and verification requirements. Besides that, the dataset could also be used for fine-tuning LLMs to generate functional coverage more accurately. Lastly, advanced prompt engineering techniques or fine-tuned models could allow the auto-evaluation framework to work on *UVM* based testbenches and generate functional coverage in *SystemVerilog*, making the work viable for all practical application.

Acknowledgment

This work was supported by Brno University of Technology under project number FIT-S-23-8141.

REFERENCES

[1] Wile, Bruce and Goss, John and Roesner, Wolfgang, *Comprehensive Functional Verification: The Complete Industry Cycle*. San Francisco, CA: Morgan Kaufmann, 2005.

[2] Bergeron, Janick, *Writing Testbenches Using SystemVerilog: Description, Tips, and Techniques*. Boston, MA: Springer, 2006.

[3] Spear, Chris and Tumbush, Greg, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd ed. Springer Publishing Company, Incorporated, 2012.

[4] Shirahatti, Anand, "High-level Specification Model based Functional Coverage Generation." [Online]. Available: https://www.verifsudha.com/2018/02/11/write-functional-coverage-plan/

[5] Ikram, Shahid and Perveiler, Jack and Akkawi, Isam and Ellis, Jim and Asher, David, "Table-based Functional Coverage Management for SoC Protocols," in *Proceedings of the Design and Verification Conference (DVCon) North America*, 2015. [Online]. Available: https://dvcon-proceedings.org/document/table-based-functional-coverage-management-for-soc-protocols/

[6] Tambekar, Nikhil, "A Practical Approach to Generating SystemVerilog Covergroups from YAML using Jinja2 Templates," in *Proceedings of the Design and Verification Conference (DVCon) Industry Track*, 2023, pp. 123–130. [Online]. Available: https://dvcon-proceedings.org/wp-content/uploads/99445.pdf

[7] Shimizu, Kanna and Dill, David L., "Deriving a Simulation Input Generator and a Coverage Metric from a Formal Specification," in *Proceedings of the International Conference on Design Automation (DAC)*. ACM/IEEE, 2002, pp. 801–806.

[8] Li, Hui and Dong, Zhen and Wang, Siao and Zhang, Hui and Shen, Liwei and Peng, Xin and She, Dongdong, "Extracting Formal Specifications From Documents Using LLMS for Test Automation," in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, 2025, pp. 1–12.

[9] El Mandouh, Eman and Wassal, Amr G., "Automatic generation of functional coverage models," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 754–757.

[10] Verma, Shireesh and Harris, Ian G. and Ramineni, Kiran, "Automatic Generation of Functional Coverage Models from Behavioral Verilog Descriptions," in *2007 Design, Automation & Test in Europe Conference & Exhibition*, 2007, pp. 1–6.

[11] Chandankhede, Ankit, "A Comprehensive Automated Generation of Functional Coverage and Structured High Verification Plan for Complex Architecture of GPUs and AI Accelerator," *Journal of Artificial Intelligence & Cloud Computing*, vol. 2, no. 4, pp. 1–4, Dec. 2023.

[12] Kumar, Pranjal, "Large language models (LLMs): survey, technical frameworks, and future challenges," *Artificial Intelligence Review*, vol. 57, no. 10, p. 260, Aug 2024. [Online]. Available: https://doi.org/10.1007/s10462-024-10888-y

[13] Murthy, Nishanth Somashekara and Nelson, Eldon and Sapatnekar, Sachin S and Sartori, John, "VerifLLMBench: An Open-Source Benchmark for Testbenches Generated with Large Language Models," in *Proceedings of the Design & Verification Conference and Exhibition (DVCon U.S.)*, 2025.

[14] Liu, Shang and Fang, Wenji and Lu, Yao and Wang, Jing and Zhang, Qijun and Zhang, Hongce and Xie, Zhiyao, "RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 4, pp. 1448–1461, 2025.

[15] Miftah, Samit Shahnawaz and Srivastava, Amisha and Kim, Hyunmin and Basu, Kanad, "Assert-O: Context-based Assertion Optimization using LLMs," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, ser. GLSVLSI '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 233–239. [Online]. Available: https://doi.org/10.1145/3649476.3660378

[16] J. Labuda, "LLM-based Functional Coverage Generation and Auto-Evaluation Framework." [Online]. Available: https://github.com/Northeus/coge/tree/DVCON2025

[17] Jyoti, Shivya and Tejpal, Moulik and R, Jothi K, "Optimizing Generative AI Applications: A Comparative Study of Effective Prompting Techniques," in *2025 5th International Conference on Pervasive Computing and Social Networking (ICPCSN)*, 2025, pp. 389–396.

[18] Snyder, Wilson and Wasson, Paul and Galbi, Duane and et al, "Verilator." [Online]. Available: https://github.com/verilator/verilator