Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions

CHUYUE SUN, Stanford University, USA
CHRISTOPHER HAHN, Stanford University, USA
CAROLINE TRIPPEL, Stanford University, USA

Assertion-based verification is a technique to ensure that a circuit design conforms to its specification and help detect errors early in the design process. It is enabled by powerful industry and open-source model-checking tools that automatically prove or disprove an assertion for a given circuit design. Formalizing a circuits requirement, however, involves a significant manual effort by verification engineers to translate requirements in natural language into a formal assertion language. In this extended abstract, we introduce a framework that utilizes Large Language Models (LLMs) pre-trained on natural language and code to improve verification productivity by automating the formalization process. In particular, we report on the current progress of developing n12sva, a framework for circuit-aware translations of natural language to the most frequently used assertion language, SystemVerilog Assertions (SVA). We introduce a methodology that (1) generates the SVA for a specific circuit out of a generic circuit property in natural language and (2) implements a model checker and human in the loop that interactively provides feedback to the verification engineer and the underlying LLM to facilitate debugging the design.

1 INTRODUCTION

Ensuring the correct operation of critical hardware components in all possible scenarios requires more than just testing. Formal verification techniques are necessary to prove or disprove, in the form of a counter-example, critical requirements on hardware designs [1, 3, 32]. Fortunately, the hardware domain is amenable to verification and constitutes a decidable problem for many practical languages (e.g., [8, 9, 14, 19, 29]). Model checking tools, such as Jaspergold [27] and Pono [17], are available to address the verification problem and scale to real-world examples.

In order to effectively apply formal verification, however, a formal specification is required that semantically captures the requirement and serves as an input to model-checking tools. Among the available specification languages, SystemVerilog Assertions (SVA) [29] is a widely used language that enables verification engineers to define complex properties, constraints, and requirements for the design under verification in a concise manner. For example, consider the following SVA that verifies the correctness of a memory write operation:

```verilog
assert property (@(posedge clk) disable iff (~reset_n) )
   (addr == 0xDEADBEEF) && (wr_en == 1'b1) &&
   (data == 32'hCAFEBABE) |->
   (mem[addr] == 32'hCAFEBABE);
```

This assertion checks that if the address is 0xDEADBEEF, the write-enable signal is high, and the input data is 32'hCAFEBABE, then the memory location at address 0xDEADBEEF should also contain the same value 32'hCAFEBABE.

Formalizing requirements given in natural language to SVA can be a time-consuming and error-prone task. It often requires manual decomposition of the requirements in order to scale to complex designs. Recent advances in deep learning have shown great potential in assisting or even outperforming humans in various natural language processing tasks, e.g., [7, 24, 33, 35], including translation [6]. Our framework leverages the abilities of deep neural networks to facilitate the translation of natural language requirements to SVA, thus reducing the burden on verification engineers and improving the quality of the formal specifications. Specifically, we propose a framework called n12sva, which utilizes Large Language Models (LLMs) (e.g., [2, 18]) to translate natural language descriptions of hardware requirements into equivalent SVA statements for specific circuit designs under verification. The framework builds on n2spec [4] a recently released tool to interactively translate natural language to temporal logics. We report on the current progress of extending prior work to the n12sva framework.

The n12sva framework provides two key contributions. First, a methodology to take the circuit design into account while translating a natural language requirement. For example generally stating that “unless reset, the output signal is assigned to the input signal” has a different meaning for different circuit designs, and needs to be instantiated accordingly: We utilize the abilities of LLMs to do in-context learning [2] and interactively adjust the prompt during the formalization and verification process. Our prompting methodology formulates the generated SVA into sub-translation for users to edit, delete, or add new entries [4]. In the future, we envision a preprocessing step that extracts key components of the circuits to provide the underlying large language models with concise information during inference, including, for example, module names, input and output wire names, and other meta information.

Second, we implement a seamless feedback loop utilizing a model checker that automatically checks the assertions on the design under verification and provides feedback to both the verification engineer and the large language model. By doing so, the engineer can adjust bugs or omissions in the SVA formalization and the LLM can attempt to debug the circuit design once the SVA captures the intended meaning.

Ultimately, n12sva aims to utilize current advances in deep learning to improve verification productivity by automatically providing circuit-aware translations to SystemVerilog Assertions.


2 BACKGROUND

We provide a brief background of SystemVerilog Assertions as the formalization language for the design’s requirements and the Large Language Models component, in our case GPT-4.

2.1 SystemVerilog Assertion (SVA)

SystemVerilog Assertions (SVA) is an expressive language extension introduced in the SystemVerilog hardware description and verification language. The primary goal of SVA is to enhance the verification process by enabling exhaustive, automated checks of the desired specifications.

SVA properties are expressed using a combination of Boolean operators and temporal operators, along with expressions and variables from the design under verification. The basic syntax of an SVA is as follows:

```plaintext
assert property_name[(property_specifier)] property_expression;
```

Where `property_name` is a unique identifier for the property, `property_specifier` is an optional specifier that can be used to specify things like clock domains or assertion severity, and `property_expression` is the actual SVA property expression.

An SVA expression is based on temporal logic [20]. The semantics of SVA is defined over an infinite execution trace, which is a sequence of states through the hardware circuit. An SVA property is true if it holds for all possible traces of the design that satisfy the constraints specified in the property.

SVA provides the standard temporal operators for specifying properties, including the following:

- `always`: asserts that a property holds for all states in a trace
- `eventually`: asserts that a property holds at some stage in a trace
- `until`: asserts that a property holds until another property becomes true
- `next (#1)`: asserts that a property holds at the next state in a trace

In addition to these temporal operators, SVA also provides Boolean operators for combining expressions, including `∧`, `∨`, `¬`, and `⊕`, as well as many more programming constructs.

2.2 Large Language Models (LLMs)

LLMs are large neural networks typically consisting of up to 176 billion parameters. They are pre-trained on massive amounts of data, such as "The Pile" [10]. Examples of LLMs include the GPT [22] and BERT [6] model families, open-source models, such as T5 [23] and Bloom [25], or commercial models, such as GPT-4 [18]. LLMs are Transformers [28], which is the state-of-the-art neural architecture for natural language processing. Additionally, Transformers have shown remarkable performance when being applied to classical problems in verification (e.g., [5, 11, 15, 26]), reasoning (e.g., [16, 36]), as well as the auto-verification [21] of mathematics and formal specifications (e.g., [12, 13, 34]).

We currently use GPT-4[18] as the large language model backend in our nl2sva framework. The framework and our prompting methodology, however, are agnostic to the underlying machine learning model. A common technique to obtain high performance with a limited amount of labeled data is so-called "few-shot prompting" [2], which we also adopted in nl2sva. The language model is presented a natural language description of the task usually accompanied by a few examples that demonstrate the input-output behavior. Our framework is based on this technique and extends the recent interactive prompting methodology in [4].

3 THE NL2SVA FRAMEWORK

3.1 Overview

Figure 1 shows an overview of the implementation of nl2sva. The framework consists of the following components. At the base of the framework lies an LLM (e.g., GPT-4 [18] or BLOOM [25]) serving as the basic translation engine to handle natural language. The LLM translates the circuit under verification and the specifications in natural language into an array of intermediate representations. An interactive engine handles these intermediate representations to provide feedback to an oracle (e.g., a developer) and automatically initiating the verification process by querying a model checker.

The intermediate representations consist of circuit metadata like input/output signals, a natural language sub-translations that a developer can leverage for debugging, and the current candidate SVA. A sub-translation is a decomposition of the requirement that maps the formalization back to parts of the natural language input. The final output of the framework is an SVA that is approved by the oracle, which is automatically passed to the model checking backend. The solid lines show the forward pass in the framework and the dotted lines show the feedback pass.

In the forward pass, users can upload their circuit designs in Verilog and provide circuit specifications (e.g., functional correctness properties, ordering properties, or security properties) in natural language. The framework automatically generates the prompt for
the LLM and passes it to the underlying LLM. After parsing the response from LLM, the framework generates the intermediate representations for the circuits, natural language sub-translations and SVA. Users can then optionally check the generated SVA directly on their circuits using the model checker.

In the feedback pass, users have the option to make edits to the natural language sub-translations and even adjust the input circuit properties. If the model checker disproves the generated SVA, users can fix their circuit designs and retry.

3.2 Demonstrative Example
A key component of nl2sva is the ability to instantiate generic circuit requirements for specific circuit designs under verification. We implemented the automatic parsing and prompting (left side of Figure 1). As a demonstrative example, we provide an initial experiment on a toy example. Extending the experiments to real-world examples is planned as a next step.

We will show how to generate SVAs on two different circuits for the same natural language circuit property. In this example, we want to translate the functional property "Unless reset, the output signal is assigned to the last input signal" on both a finite state machine shown in Appendix A.1 and a D-Flip-Flop shown in Appendix A.2. Note that even with the same natural language input, the two output SVAs are considerably different.

For the finite state machine circuit, we received the following:

```verbatim
1 assert property @posedge clk
2 (if (!reset) valid == $past(c));
```

Listing 1. FSM translation

For the D-Flip-Flop circuit, we received the following:

```verbatim
1 assert property @(posedge clk)
2 (!async_reset) |-> (Q == $past(D));
```

Listing 2. DFF translation

We inserted the above FSM translation 1 at the end of the FSM circuit A.1 and the above DFF translation 2 at the end of the DFF circuit A.2 right before endmodule. A model checking tool can then be run on the design including the inserted assertion. As a next step in the development of the nl2sva framework, we plan to incorporate the automatic SVA checking, which is currently under development. We ran JasperGold manually, with both assertions being proven.

From our experiments, we observe that the LLM clearly is capable of instantiating the toy generic natural language requirement to specific toy circuit designs. For example, it automatically maps "reset" from the natural language circuit property input to reset signal in the FSM circuit and async_reset signal in the DFF circuit. It also clearly knows that the output signal is valid in FSM and Q in DFF. The language model is also able to write syntactically correct SVA without any tutorial (as it has been extensively trained on it) and in our case, the translation is also semantically correct. However, due to the inherent nondeterminism in LLMs, it produces different syntax for two circuits, although both are semantically correct.

3.3 Implementation and Challenges
The framework is implemented in Python 3 and flask framework [30]. By default, we use GPT4 [18] as the LLM and JasperGold [27] as the model checker, but the framework is agnostic to the underlying tools. We extended the frontend of [4] to handle the human feedback. The frontend web interface (see Figure 2 in the appendix) has four important elements: "Prompt", "Circuit in Verilog", "Subtranslations", and "Final Result". The tool takes a natural language circuit property as input and output concrete SVA under the context of specific circuit designs along with sub-translations. Users can also optionally add sub-translations and adjust model hyper-parameters (model temperature and the number of sampling tries). To take in human feedback, users can edit, delete and add sub-translations from the frontend.

Currently under development is providing the model checker feedback to the language model and the user. The backend handles prompt generation, API calls to the LLM, and post-processing of LLM feedback, i.e., selecting the most promising translation based on model confidence score.

To generate high-quality SVA translations, we use the "few-shot prompting" [2] technique. The body of the prompt consists of a fixed prompt and an interactive prompt. The minimal fixed prompt shown in Appendix B includes only one simple circuit design and four translation examples. And the interactive prompt includes the user-uploaded circuit, natural language circuit property, and the optional sub-translation. In the few-shot examples, we adopt the "chain-of-thought" [31] technique to help LLM reason. The purpose of the fixed prompt is to show LLM how to produce a useful response in the format we expect. Hence, we append the interactive prompt to the end of the fixed prompt so we can expect the LLM to fill in the translated SVA the same way we do in the fixed prompt. In our case, we can expect the LLM-transported SVA to be right after So the final SVA translation is and end with the FINISH token.

We are susceptible to outside computational resources and API limitations. For example, the default GPT-4 API only supports up to 8192 tokens of context memory. This means that for complex circuit designs, we have to manually decompose large circuits into smaller independent modules to feed into the framework. For future work, we plan to implement a preprocessing step that automatically extracts only the necessary information of the circuit for the LLM to succeed in the translation task. We plan to conduct more experiments on real-world circuit designs and collect more feedback from the framework users. To enhance the generated SVA quality, we will continue to improve our prompting techniques and even formulate the model checker generated counter-examples to feedback to the LLM.

4 CONCLUSION
In this extended abstract, we have introduced nl2sva, a framework that enables the translation of natural language specifications to SystemVerilog Assertions (SVA). We have provided an overview of the current state of development, described its implementation and highlighted the current challenges, especially handling large circuit designs.


REFERENCES


[26] A.1 Finite State Machine

```
module fsm_example(  
    input clk,  
    input reset,  
    input c,  
    output valid  
);  
    parameter [3:0]  
        idle = 3’d0,  
        one_str = 3’d1,  
        zero_str = 3’d2,  
        invalid_str = 3’d3;  
    reg [2:0] state, nxt_state;  
    always @(c or state or reset) begin  
        if (reset) begin  
            nxt_state = idle;  
            valid = 0;  
        end else begin  
            valid = 0;  
            case(state)  
                idle: if (c) begin  
                    nxt_state = one_str;  
                end else begin  
                    nxt_state = idle;  
                end  
                one_str: if (c) begin  
                    nxt_state = one_str;  
                end else begin  
                    nxt_state = zero_str;  
                end  
                zero_str: if (c) begin  
```
nxt_state = valid_str;
end else begin
  nxt_state = zero_str;
end
valid_str: begin
  if (c)
    nxt_state = valid_str;
  else
    nxt_state = invalid_str;
end
valid = 1;
end
invalid_str: begin
  nxt_state = invalid_str;
  valid = 0;
end
default: nxt_state = 3’bx;
endcase
always @(posedge clk) begin
  state <= nxt_state;
end
endmodule

A.2 D-Flip-Flop
module RisingEdge_DFlipFlop_AsyncResetHigh (D, clk, async_reset, Q);
input D; // Data input
input clk; // clock input
input async_reset; // asynchronous reset high level
output reg Q; // output Q
always @(posedge clk or posedge async_reset) begin
  if(async_reset==1'b1)
    Q <= 1'b0;
  else
    Q <= D;
end
endmodule

B PROMPT
Following is the design for tff:
module tff(
  input wire clk,
  input wire reset,
  input wire T,
  output wire Q
);
reg Q_reg;
always @(posedge clk or posedge reset) begin
  if (reset) begin
    Q_reg <= 1'b0;
  end else begin
    if (T) begin
      Q_reg <= ~Q_reg;
    end
  end
  assign Q = Q_reg;
endmodule

Natural Language: on falling clock ticks, if reset is true then output is true in the next one or two cycles.
Explanation: "reset" from the input translates to the atomic proposition restn in the tff module and "output" translates to the atomic proposition Q in the tff module. The clock tick is the atomic proposition clk in the tff module.
Explanation: ##[1:2] Q means that Q is true on the next clock, or on the one following (or both). |-> is the implication operator, so this assertion checks that whenever restn is asserted, Q must be asserted on the next clock, or the following clock.
"on falling clock edge" translates to @(negedge clk).
Explanation dictionary: {"on falling clock ticks": @(negedge clk), "if ... then ... ": "|->", in the next one or two cycles": "##[1:2]", "reset":"restn", "output":"q"}
So the final SVA translation is assert property (@(negedge clk) restn |-> ##[1:2] Q). FINISH

Natural Language: all inputs are never true at the same time during any point of simulation.
Explanation: there are three inputs: clk, reset, T. There is one ouput: Q.
Explanation dictionary: {"all three inputs": "clk && reset && T", "never true": "!"}
So the final SVA translation is assert property (! (clk && reset && T)).FINISH

Natural Language: at least two inputs are true at the same time during any point of simulation.
Explanation dictionary: {"least two inputs": " (clk && reset) || (reset && T) || (clk && T)"}
So the final SVA translation is assert property ((clk && reset) || (reset && T) || (clk && T)).FINISH

Natural Language: The circuit output is always valid.
Explanation: The circuit is valid when the output \( Q \) either remains the same or toggles when the input \( T \) is high during a rising edge of the clock.

So the final SVA translation is assert property @posedge clk \( (T == 1'b1) \rightarrow (Q\_reg == \sim \$past(Q\_reg)) \#1 (T == 1'b0) \rightarrow (Q\_reg == $past(Q\_reg)). \text{FINISH} \)