Using LLMs to Facilitate Formal Verification of RTL

Marcelo Orenes-Vera, Margaret Martonosi and David Wentzlaff
Department of Computer Science and Electrical Engineering, Princeton University
Princeton, New Jersey, USA
Email: {movera, mrm, wentzlaf}@princeton.edu

***Abstract*—Formal property verification (FPV) has existed for decades and has been shown to be effective at finding intricate RTL bugs. However, formal properties, such as those written as SystemVerilog Assertions (SVA), are time-consuming and error-prone to write, even for experienced users. Prior work has attempted to lighten this burden by raising the abstraction level so that SVA is generated from high-level specifications. However, this does not eliminate the manual effort of reasoning and writing about the detailed hardware behavior. Motivated by the increased need for FPV in the era of heterogeneous hardware and the advances in large language models (LLMs), we set out to explore whether LLMs can capture RTL behavior and generate correct SVA properties. First, we design an FPV-based evaluation framework that measures the correctness and completeness of SVA. Then, we evaluate GPT4 iteratively to craft the set of syntax and semantic rules needed to prompt it toward creating better SVA. We extend the open-source AutoSVA framework by integrating our improved GPT4-based flow to generate safety properties, in addition to facilitating their existing flow for liveness properties. Lastly, our use cases evaluate (1) the FPV coverage of GPT4-generated SVA on complex open-source RTL and (2) using generated SVA to prompt GPT4 to create RTL from scratch. Through these experiments, we find that GPT4 can generate correct SVA even for flawed RTL—without mirroring design errors. Particularly, it generated SVA that exposed a bug in the RISC-V CVA6 core that eluded the prior work's evaluation.**

I. Introduction and Background

The Cambrian explosion in the diversity of hardware caused by the end of Moore's law has exacerbated the challenges associated with RTL design verification (DV) [12]. Formal property verification (FPV) utilizing industry-standard SystemVerilog Assertions (SVA) [6] is becoming increasingly important to provide exhaustive DV in the face of growing hardware complexity and variety. SVA can express temporal relations over RTL signals, which fall into two major classes: safety and liveness properties. Safety specifies that *"nothing bad will happen"*, e.g., FSM transitions, while liveness specifies that *"something good will happen"*, e.g., a request should eventually get a response. FPV tools [3], [15] use solver engines based on formal methods to search for counterexamples (CEXs) exhaustively. While FPV is very effective for DV, engineers often feel discouraged from using it because of the steep learning curve and additional effort to write assertions [13].

Prior work has tried to ease the use of FPV by automating parts of the process: AutoSVA [11] generates end-to-end liveness properties from an annotated RTL module interface; ILA [5] generates a model of the design from a functional specification and compares it against its RTL implementation; and RTLCheck [8] verifies the RTL of CPU pipelines for memory consistency by synthesizing SVA from axiomatic specifications.

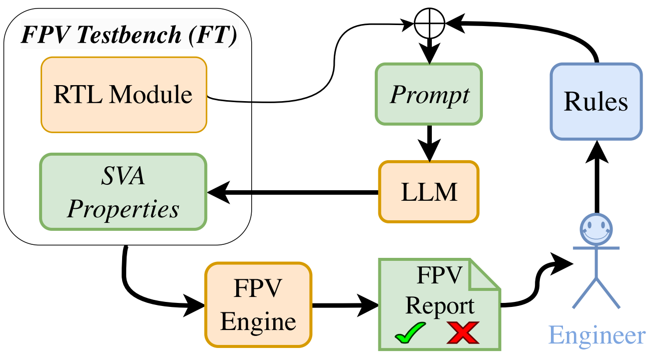

Fig. 1. FPV-based evaluation framework. The FPV tool returns whether the assertions generated by the LLM are correct or not—for a given RTL. Hinted by the errors or CEXs of the FPV report, the engineer manually writes or refines the rules that guide the LLM toward generating better SVA. The rule set and the RTL are combined into a prompt in order to generate a new iteration of SVA properties. The green boxes are automated steps; the blue ones are manual.

While these are effective tools, they either verify subsets of RTL designs or require significant effort to write structured specifications. With the recent advances in LLMs, a question arises: Can LLMs help accelerate RTL design and verification? And if so, how should we integrate them into modern RTL development? In the last few months, researchers have explored using LLMs to generate temporal logic specifications and assertions from natural language [4], [7], as well as filling gaps in incomplete RTL [14]. We take a more holistic approach and **explore whether LLMs can generate correct SVA** for a given design **without any specification beyond the RTL**—even when the RTL contains bugs. Our **motivation** for that is that specifications are not always available or precise enough. Often, implementation details are not fleshed out until the RTL is written. This is especially true in academic and open-source hardware, where the RTL is in continuous development [10]. Moreover, generating SVA solely from RTL would enable formally verifying RTL that has been generated by LLMs [14].

Our approach: Starting with an empty FPV testbench (FT) for an RTL module, we aim to generate SVA that reasons about the correctness of the design without needing to manually provide details about functionality or the properties to be generated. We utilize GPT4 [9] for this holistic task because early experiments with smaller LLMs were not promising. However, even state-of-the-art GPT4 generates syntactically and semantically wrong SVA by default. Thus, we first had to *teach* GPT4 how to generate correct SVA by iteratively refining the prompt with rules (Fig. 1). We then build on top of

AutoSVA [11] to include our improved GPT4-based flow for safety properties in addition to facilitating the existing liveness flow (Fig. 2). We make our extended framework available—anonymized for now—as AutoSVA2 [2]. Our use cases evaluate (1) the FPV coverage of AutoSVA2 on complex RTL, and (2) using generated SVA to prompt GPT4 to create RTL from scratch, for which AutoSVA2 can output more SVA (Fig. 3).

Our technical contributions are:

- An iterative methodology based on FPV to find the rules required to *teach* an LLM how to generate syntactically and semantically correct SVA from a given RTL module.
- Evaluating GPT4 and crafting the rules that improve its SVA output, and extending the AutoSVA framework with this improved GPT4-based SVA generation flow.
- Characterizing robustness and coverage of AutoSVA2.
- An AutoSVA2-driven RTL generation and verification methodology; iteratively improves LLM-generated RTL by prompting human-refined generated SVA (Fig. 3).

Our experiments found that:

- GPT4's creativity allows it to generate correct SVA from buggy RTL, i.e., it is not compelled to generate SVA solely based on the RTL we have provided.
- GPT4 is not significantly sensitive to the names of the RTL module and variables in order to generate SVA.
- For the same RTL modules, AutoSVA2-generated properties improved coverage of RTL behavior by up to $6\times$ over AutoSVA-generated ones. [1].
- Within an hour of engineering effort, our AutoSVA2 evaluation exposed a bug in the RISC-V CVA6 Ariane core [10] that eluded AutoSVA's prior evaluation [11].
- GPT4 generates better RTL when we include SVA in the prompt; its creativity allows it to generate correct RTL even if the SVA was not entirely correct.

The rest of the paper is organized as follows: Sec. II shows our iterative approach to find flaws in the LLM-generated SVA and test rules to steer it toward generating better SVA. Sec. III introduces AutoSVA2, our extended framework that integrates a GPT4-based flow on top of AutoSVA to create more complete FTs with less effort. Sec. IV and V present our two uses cases.

II. Iteratively Teaching SVA to LLMs

Our early experiments with GPT4 showed us that it could generate several SVA properties solely from RTL, but they contained syntactic and semantic errors. However, we found that we could nudge GPT4 towards generating better SVA by giving it rules to follow. Sec. II-A describes the methodology we used to iteratively construct the set of rules that are needed in the prompt for GPT4 to generate useful assertions. Sec. II-B describes the issues we encountered and the rules we added to the prompt to overcome them.

A. Rule-refinement Methodology

Fig. 1 depicts the evaluation framework we use to iteratively refine the set of rules to be included in the prompt of an LLM. Although all our experiments with this framework have been done on GPT4 (Table I), we argue that it can be used to assess the output quality of any LLM.

This methodology requires having an FT. We can create one quickly by executing the AutoSVA [1] script indicating the target RTL module. (The generated FT has property and tool-binding files but no assertions.) In theory, one could use any RTL module as input to the LLM, but note that the engineer should be able to easily determine the issues with the SVA. Ideally, the RTL module used should be entirely correct, so that the CEXs generated by the FPV engine are due to wrong assertions. Recall that the goal of this methodology is not to provide a complete FT for this RTL module but rather to refine the rules for the LLM to generate better SVA. For our experiments with GPT4 (detailed in Sec. II-B), we used the FIFO module from the AutoSVA repository [1].

B. Experiments with GPT4

We apply the above methodology to GPT4 to test its generated SVA and assess whether our hand-written rules improve it. Particularly, we used the 8K-token version offered via OpenAI's chat interface [9]. We use a clean-slate context for each iteration to avoid polluting the new SVA with previous issues.

Reproducibility is challenging with LLMs because of their creativity—a key attribute of our interest in LLMs for SVA generation. Our best effort towards reproducibility is to be fully transparent with our experiments. To do that, we made a separate commit for each iteration on our anonymized repository [2] starting from a fork of AutoSVA [1]. This repository includes prompt, response, and FT for each iteration and a booklog with our observations for all the experiments described in this paper.

Engineering effort: It took 23 iterations and $\sim$8 hours[1] to create the rules for GPT4 to output a complete and correct set of assertions for the FIFO module. Correctness was shown by full proof, and completeness was shown from statement and toggle coverage. We obtained assertion validity and coverage using JasperGold (JG) [3]. JG took just a few seconds to compile and test the assertions on our server. GPT4 generated each set of SVA in under a minute. Most of the time was spent auditing the assertions and carefully writing and refining the rules.

LLM cost: We already had a monthly subscription to this model, so these experiments did not incur extra costs. However, note that when used via the API, queries to this model currently cost 0.03 USD per 1000 tokens, which again incentivizes using a small RTL module for this task.

Table I shows, for each iteration test (T), whether the FPV tool successfully compiled the property file, the number of assertions generated and failing, and the main issues found.[2]

We group the issues into four categories: not compiling due to wrongly referencing internal signals (IN) or wrong syntax (SY); and compiling but using wrong timing (WT) or wrong semantics (WS). For tests where JG could not successfully compile the FT (✗), we did not attempt to fix it to check how many assertions failed (−), except on a few iterations when the compilation error was minor (✗).

[1] The engineering time can be observed from the commit timestamps [2]

[2] The booklog contains details of the issues observed at each iteration [2]

TABLE I
STATUS FOR COMPILATION AND SVA CORRECTNESS (# OF PROPERTIES GENERATED AND FAILING) FOR EACH ITERATION TEST (T).

T	Compile	#Prop	#Fail	**Main issues**
1	✗	4	–	IN: Undeclared var (no module prefix)
2	✗	6	–	SY: Wrong keyword for assertion
3	✗	8	–	SY: Using *foreach* as an assertion loop
4	✗	4	–	IN: Undeclared *buffer_head_r*
5	✗	9	–	SY: Error in include and assert naming
6	✗	6	–	SY: Duplicated assertion name
7	✓	6	4	WT: Wrong time semantics $\mid\!-\,>$
8	✗	6	3	IN: Undeclared var (no module prefix)
9	✗	5	–	IN: Module prefix; ignored prev. rules
10	✓	9	5	WT: Wrong time semantics $\mid =>$
11	✓	7	1	WT: Missing $past in postcondition
12	✓	10	7	WT: Too much $past usage
13	✗	9	–	SY: Forgot *foreach* rule from T3
14	✓	12	4	WS: Incr. without wrap; wrong signal
15	✓	8	2	WT: Missing $past in postcondition
16	✓	9	4	WT/WS: Wrong bitwise manipulation
17	✓	8	3	WT: Wrong time semantics $\mid =>$
18	✗	10	0	SY: Array-named assertions as_name[i]
19	✗	12	2	SY/WS: Empty precond.;wrong bitwise
20	✗	10	–	SY: Wrong width in constant usage
21	✓	7	1	WT: Missing $past for register
22	✓	9	1	WT: Incorrect $past in precondition
23	✓	8	0	Full Proof
24	✓	8	1	Assuming wrong behavior about *out_rdy*

Internal Signals (IN): Since our first iteration, we observed that GPT4 would not properly reference internal signals (not declared in the module interface). It took several iterations to refine the rules for GPT4 to use hierarchical referencing by prefixing the module name before the internal signal. These rules are shown in lines 4-5 of Listing 2.

Syntax (SY): We found errors related to using incorrect keywords, e.g., `always@(<condition>)` instead of `assert`, and wrong module include and property names. GPT4 also kept using *foreach* wrongly to create loops of assertions. Our rules to fix syntax issues are described in lines 1-3.

Wrong Timing (WT): One of the hardest issues to fix was the concept of time in SVA; e.g., GPT4 kept using same-cycle implications ($\mid\!-\,>$) for reasoning about the updated value of registers (flip-flops). Timing becomes especially problematic when updating registers within an array because the index selector may also be a register. In this case, the postcondition should reason about the updated value of the register array but the old value of the index. We show an example of this from T16 in Listing 1; the assertion should have used `$past` for `buffer_head_r`. Our rules to overcome that (lines 9-16) include teaching the concept of registers and combinational logic, same- and next-cycle assertions, and when to use `$past`.

Wrong Semantics (WS): Beyond timing, we found other wrong semantics in counter increment logic, `$countones` and bitwise operators. Listing 1 shows an example of this, where the comment generated by GPT4 is correct, but the assertion is wrongly using `!&` instead of `!|` to check that all buffer slots are invalid. We fixed them by adding rules to teach GPT4 about the correct behavior of these operators (lines 6-8).

```
// Check that if there's an in handshake (in_hsk), the
    buffer corresponding to the head of the FIFO should be
    updated with data on the next cycle.
as__in_hsk_data_update: assert property (fifo.in_hsk |=>
    fifo.buffer_data_r[fifo.buffer_head_r]==$past(in_data));

// Check that out_val should be low if all slots are invalid
as__out_val_low_if_all_buffer_invalid:  assert property
    ((!&fifo.buffer_val_r) |-> fifo.out_val == 0'b0);
```

Listing 1. Semantically wrong assertions from iteration test 16.

```
1 DO NOT declare properties; DECLARE assertions named as__<
      NAME>: assert property (<EXPRESSION>).
2 DO NOT use [] at the end of assertion NAME. Do not add @(
      posedge clk) to EXPRESSION.
3 DO NOT use foreach loops in assertions, use generate for.
4 Internal signals are those NOT present in the interface.
      Internal signals are declared within the module.
5 Referencing internal signals in the property file ALWAYS
      requires prepending the name of the module before the
      signal name, e.g., name.<internal_signal>.
6 &bitarray means that ALL the bits are ONES.
7 !(&bitarray) means it's NOT TRUE that ALL the bits are ONES
      , i.e., SOME of the bits are ZEROS.
8 !(|bitarray) means that NONE of the bits are ONES, i.e.,
      ALL the bits are ZEROS.
9 Signals ending in _reg are registers: the assigned value
      changes in the next cycle.
10 Signals NOT ending in _reg are wires: the assigned value
      changes in the same cycle.
11 USE a same-cycle assertion (|->) to reason about behavior
      occurring in the same cycle.
12 USE a next-cycle assertion (|=>) to reason about behavior
      occurring in the next cycle, for example, the updated
      value of a _reg.
13 DO NOT USE $past() in preconditions, ONLY in postconditions
14 DO NOT USE $past() on postcondition of same-cycle assertion
15 On the postcondition of next-cycle assertions (|=>), USE
      $past() to refer to the value of wires or a _reg on
      the cycle of the precondition.
16 On the postcondition of next-cycle assertions (|=>), DO NOT
       USE $past() to refer to the updated value of _reg.
```

Listing 2. Some of the rules we crafted to refine GPT4's SVA generation—written in plain (imperative) English based on our knowledge of SVA.

C. Assertion Quality and Determinism

It took 7 iterations to get the first correct syntax and 18 iterations for all assertions to pass, but we only first achieved correct syntax and full proof at T23.

Creativity: We made an extra iteration to check for output determinism given the same input. Of the eight assertions of T24, only two were the same as T23, and the rest were slight variations to test similar behavior. Listing 3 shows two of the variations that check data integrity.

```
// T23: If FIFO is not empty and output handshake is true,
    the data at 'buffer_tail_reg' index should be sent out
    in the same cycle
for (genvar i=0; i<INFLIGHT; i=i+1) begin: check_data_on_wr
   as__data_correctly_written: assert property (fifo.in_hsk
    && !(|fifo.buffer_val_reg) && (fifo.buffer_head_reg == i
    ) |=> fifo.buffer_data_reg[i] == $past(fifo.in_data));
end

// T24: When add_buffer flag is set, the next value of
    buffer_data_reg at that index should be in_data.
for (genvar i=0; i<INFLIGHT; i=i+1) begin: check_buffer_data
  as__buffer_data_set: assert property (fifo.add_buffer[i]
    |=> fifo.buffer_data_reg[i] == $past(fifo.in_data));
end
```

Listing 3. Assertions from T23 and 24 to check data integrity. The first one has a larger scope, since it uses more primary signals while the second one uses an intermidiate signal from the design (add_buffer).

Asserting unknown behaviors: T24 had a failing assertion that checked that if the FIFO consumer is not ready to read (`out_rdy` low), it must be because the FIFO is empty. Since `out_rdy` is an input to the FIFO, the FPV can set it to any value at any given time, and thus the assertion fails. However, we found it interesting that GPT4 tried to assert the behavior of a signal that was not driven inside the RTL.

RTL coverage: With the refinement of rules, GPT4 was encouraged to use more features, and with that, it generated more assertions. We also attribute that increase in assertion count to the better utilization of the context size. After T9, we instructed GPT4 to output only assertions and comments, but not module interfaces or further explanations, to save tokens in favor of producing better SVA. (Sec. III elaborates more on how to use the context efficiently.) Via JG's coverage tool, we found that the SVA from T23 and T24 achieved full coverage of the FIFO module. Although a FIFO is not a complex module, note that the SVA was generated from scratch solely based on the RTL code and the generic set of rules from Listing 2. Sec. IV evaluates more complex RTL modules and shows how generating multiple batches of SVA increases coverage.

Robustness: We made two more tests to check the robustness of GPT4 with respect to signal names. For T25, we replaced `fifo` with `modul` throughout the entire RTL module, including the module name, to discern whether prior knowledge about FIFO behavior enhanced GPT4's output. GPT4 generated a set of assertions with similar quality as T24. For T26, we replaced `modul` with `multiplier` to investigate whether prior knowledge about multipliers would worsen GPT4's output for the same RTL behavior. Again, we found no significant impact. We also observe no sign of GPT4 internally learning from prompting the rules repeatedly since removing the rule set results in a similar SVA as in T1.

Rule set completeness: The fact that our set of rules was good enough for our FIFO module does not mean it is sufficient for every module. Several of the SVA features not encountered during our rule-refinement experiments could still cause trouble for GPT4. For example, `$countones` only appeared at T25—wrongly used—so we needed to add an extra rule. This is to say, as more RTL modules are tested with GPT4, rules may need to be appended. Moreover, particular strategies could be used to nudge GPT4 to generate assertions in a particular way, e.g., to write assertions about FSMs transitions (tested in Sec. IV-A).

III. AutoSVA2: Extending AutoSVA with GPT4

The above experiments show that SVA generated by GPT4 is useful, not because it's perfect (which is not), but because GPT4 generates SVA that checks behaviors beyond what is written in the RTL—often spanning multiple assignment steps.

We decided to integrate our new SVA generation flow into AutoSVA because: (1) AutoSVA already generates the FT scaffolding we need to run SVA properties on FPV tools like JasperGold (JG) [3] and YosysHQ's SBY [15]. (2) properties generated with AutoSVA and GPT4 can be complementary (Sec. IV-B); (3) AutoSVA is open-source and continues being extended for new features, e.g., hardware security [12].

Fig. 2 depicts how AutoSVA2—by combining our new flow (green arrows) with the existing one from AutoSVA—creates a more complete FT We also added an extra flow (blue arrows) to lighten the effort of adding annotations to the RTL module

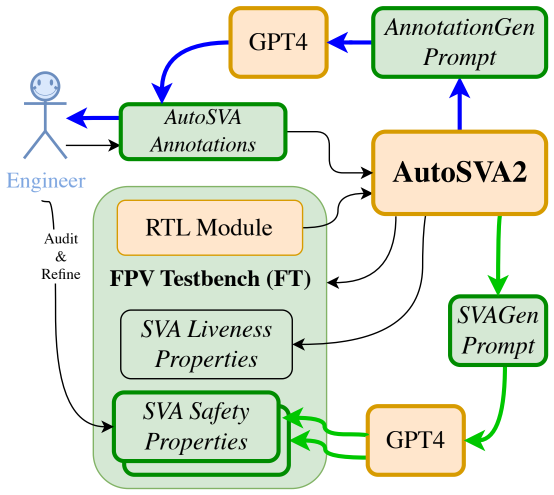

Fig. 2. Overview of AutoSVA2. Our additions to the original AutoSVA flow are shown with thick boxes and arrows; the original flow is shown with thin boxes and arrows. The green boxes indicate automatically generated artifacts. The green arrows indicate the SVA generation flow and the blue arrows the annotation generation flow. The engineer in the loop revises both the annotations and the generated SVA.

interface. The engineer audits the generated annotations and SVA and corrects them if necessary.

Interface-annotation flow: We crafted another set of rules[3] to teach GPT4 how to generate the annotations about module transactions that the AutoSVA paper [11] introduced in order to generate end-to-end liveness properties. As depicted in Fig. 2, AutoSVA2 takes the RTL module as input and appends these rules to compose a prompt for GPT4. We found this flow to capture transactions correctly on RTL components with clear interfaces like the FIFO, with valid syntax and semantics. However, for more complex modules with several interfaces like those evaluated in Sec. IV, it grouped together interfaces and signals that are not part of the same transaction, e.g., the memory request interface and the response into the TLB. We argue that even when the annotations are incorrect, it is still easier for an engineer to correct them than starting from scratch.

Completeness of the FT: The engineer may want to generate multiple batches of SVA, as we found that to increase completeness (Sec. IV-B) and having redundant assertions is not problematic for FPV tools—they can reuse the same state space exploration to prove multiple properties. On the contrary, the annotation flow is only triggered once when the FT is first created. The coverage metrics reported by FPV tools like JG can guide the engineer on which interface signals may be missing annotations and which other internal signals miss assertions. Future work could extract from the FPV report the RTL lines that are not being covered and use that to generate a more targeted prompt—until the SVA covers the entire design.

Completeness vs Correctness: Having SVA with full RTL coverage does not imply that the assertions or the RTL are correct. (a) assertions producing CEXs still contribute to coverage, and (b) assertions may have the same bug as the RTL. Our goal with AutoSVA2 is that the generated assertions have as much coverage as possible so that the engineer does not need to come

[3]Included in our anonymized GitHub repository [2]

up with new properties but rather focus on auditing the existing one to ensure it matches the hardware specification. Note that the engineer must audit all assertions, not only those producing CEXs, as they may be proving the wrong thing. Throughout our experiments, **we found GPT4's creativity valuable for SVA generation:** the fact that it generates similar properties with slight variations makes it prone to create buggy SVA for correct RTL but also to create correct SVA for buggy RTL.

Filtering comments from RTL design: It may seem counterintuitive at first, but we found that removing comments from the RTL design improves the quality of the generated SVA for larger RTL modules (e.g., those tested Sec. IV). The GPT4 model we use has a context of 8K tokens for input and output combined; when the prompt gets close to that limit, the model start forgetting part of the input in order to generate the output.

Larger context sizes vs Modularity: If the module contains too many tokens even when comments are removed (>500 lines), we see two options to move forward: (1) use a model with a larger context size, e.g., GPT4-32K, or (2) break down the RTL into smaller modules. We advocate for the latter, not only because the 32K version currently costs twice as much as the 8K one per token via the API but because it is good coding practice to create submodules for self-contained functionality.

IV. Use case 1: Testing AutoSVA2 on Complex RTL

We applied our new SVA flow with the page-table walker (PTW) and translation look-aside buffer (TLB) of the 64-bit RISC-V CVA6 Ariane core. We chose these modules for their complexity and because they were also evaluated in the original AutoSVA paper [11], so we can compare the results.

Goals: With these experiments, we are not aiming to return fully verified RTL—auditing the CEXs would require a functional specification or more knowledge about the design. Instead, we aim to test our GPT4-based SVA generation flow on complex RTL modules and compare the coverage of the new properties over the existing ones from AutoSVA.

A. Building the PTW FT and exposing an RTL Bug

Because CVA6 is widely used in the open-hardware community to build chips, it keeps being actively developed. We found that the PTW code evaluated by AutoSVA has changed since then. Tracking the bug fixes in the OpenHW Group's CVA6 repo, we found that the PTW had a bug that was fixed recently.[4] This bug was not uncovered by the assertions generated in AutoSVA's evaluation [11]. Thus, we set out to evaluate whether AutoSVA2 could find this bug.

We started by rebuilding the PTW FT from the AutoSVA repository [1]. Within a few minutes[1] we had generated the first batch[5] of 12 assertions. We generated two more batches for a total of 36 assertions. After spending half an hour auditing the assertions and the RTL, we found one failing assertion that, if refined properly, could uncover the bug.

We kept generating more assertions to find whether GPT4 could generate the correct assertions that would fail because of the RTL bug; it did after six batches—totaling 80 assertions.[3]

[4]https://github.com/openhwgroup/cva6/pull/1184

[5]We call **batch** to the output from prompting GPT4 into generating SVA.

```
// When in WAIT_RVALID, should remain as it was in the
     previous cycle if not ptw.data_rvalid_q
asgpt__wait_rvalid_tag_valid_stable: assert property (
    (ptw.state_q == WAIT_RVALID) && !ptw.data_rvalid_q |=>
     ptw.state_q == WAIT_RVALID);
```

Listing 4. Assertion uncovering the PTW bug.

Listing 4 shows the failing assertion, which actually checks the correct transition for the PTW's FSM according to the bugfix commit.[4] Once we then applied the fix,[4] the assertion proved in a few seconds of FPV tool runtime.

Assertion generation strategy: The literature on formal verification describes different strategies to write assertions [13]. For FSMs, that strategy can be creating assertions for each state transition. We appended this to the prompt to instruct GPT4 to assert when FSMs change or retain states. Other strategies could potentially be added to nudge GPT4 in a certain direction, e.g., to generate assertions for output signals based on intermediate signals and continue backward toward the inputs.

B. RTL Coverage of Automatically Generated SVA

We evaluated statement and toggle coverage for PTW and TLB with different sets of assertions: the assertions from the AutoSVA evaluation [1]; one, three, and six batches of GPT4-generated assertions; and all assertions combined.

Multi-batch improvements: We studied coverage improvement, starting with the existing AutoSVA assertions and adding batches of AutoSVA2 outputs. For PTW, we obtained a 1.05×, 1.23×, and 1.25× increase in statement coverage with one, three, and six batches, respectively, over AutoSVA assertions alone; and 1.24×, 1.57×, and 1.57× increase in toggle coverage. For TLB, the improvements are much larger: 2×, 6×, and 6× increase in statement coverage; and 2.12×, 3.67×, and 3.74× increase in toggle coverage. These results show that (a) AutoSVA2 improves coverage significantly over AutoSVA and (b) it is worth generating multiple batches, although we observed little to no improvements after three batches.

Complementary assertions: The assertions generated by GPT4 and AutoSVA are sometimes complementary, covering different parts of the design. Although for the TLB we observed that having the AutoSVA assertions did not affect the final coverage, for PTW, the combination of AutoSVA and GPT4 assertions had 1.59× and 1.34× more statement and toggle coverage, respectively, over the GPT4 batches alone. This makes sense because AutoSVA generates end-to-end properties for interface transactions, while GPT4 mostly generates assertions for internal behavior. It is hard for GPT4 to generate end-to-end properties because they are not directly observable from the RTL. Even for cases like the TLB where the coverage is overlapped, it is often easier for FPV tools to prove end-to-end assertions out of smaller assertions [13].

V. Use case 2: AutoSVA2 guiding RTL generation

LLMs have been previously used to generate RTL in partially incomplete modules [14]. For the holistic task of generating RTL from scratch, we set out to evaluate whether prompting GPT4 with SVA can help it generate better RTL.

Fig. 3 depicts the flow we devise to generate RTL from SVA (blue arrows) in addition to the AutoSVA2 flow for generating

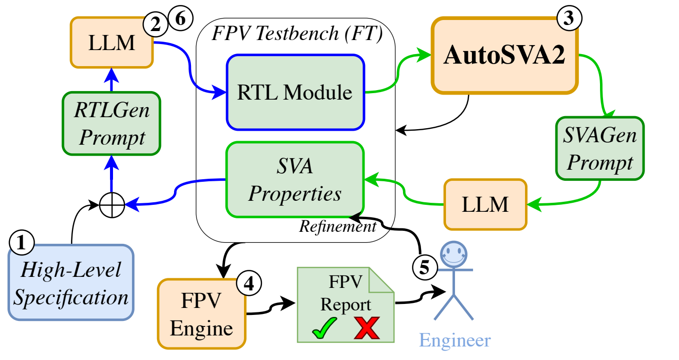

Fig. 3. Methodology for iteratively building RTL and FT. The blue arrows show the RTL generation flow; green arrows, the SVA flow; black arrows, the assertion debugging flow with the human in the loop. The green boxes indicate they are automatically generated, and the blue box, manually written.

SVA from RTL (green arrows): ① Start with a high-level specification in English; ② The LLM generates a first version of the RTL based on the specification, the module interface, and an order to generate synthesizable Verilog; ③ AutoSVA2 generates an FT based on the RTL; ④ JasperGold evaluates the FT; ⑤ The engineer audits and fixes the SVA; ⑥ The LLM generates a new version of the RTL after appending the SVA to the previous prompt. Steps ③ to ⑥ are then repeated until ***convergence***: either (a) full proof and coverage of the FT or (b) a plateau in the improvements of the RTL and SVA.

Our experiment: We used this flow to generate a FIFO queue starting with a specification of $\sim$50 words[3]). We achieved convergence by full proof after two RTL iterations. We discuss here our observations from this experiment.

SVA from the first RTL: From the FPV report and the SVA, we observed that 5 out of 11 assertions failed due to SVA issues; we made minor fixes in three of them (similar to T22 from Table I), a partial rewrite in another one, and directly removed one that was not salvageable. Interestingly, we found a valid assertion that failed due to a wrong RTL implementation and an assertion that failed due to issues in both RTL and SVA regarding the empty/full flags of the FIFO.[3] We fixed that assertion but not the RTL since (as shown in Fig. 3) we do not prompt the old RTL to GPT4 to generate the next RTL version.

RTL from the refined SVA: We found the RTL generated from the refined SVA to be much better than the first version; not only did it not have the bug, but the RTL was also more readable. The assertions for the full/empty flags were still failing; we found via the CEXs that the write pointer was missing the bit selection (to ignore the carry bit) while comparing it with the read pointer. After fixing this on the RTL, the assertion kept failing. We observed that we had wrongly specified the empty/full flags on the manually-revised assertion from the previous step. Once we fixed that all assertions proved.

Takeaways: From this use case, we conclude that (a) this iterative methodology is an efficient and effective way to bring up RTL and FTs from scratch, by having the verification engineer in the loop reviewing and fixing the SVA; (b) that errors in the RTL do not preclude GPT4 from generating correct SVA; (c) that even if the engineer erroneously modifies the SVA, GPT4 can still generate correct RTL.

VI. Discussion and Conclusion

While FPV techniques have been around for decades and are acknowledged as the most exhaustive method for DV, they are also exhausting for engineers to apply. Prior work lightens this burden by raising the level of abstraction and generating SVA from high-level specifications [5], [11]. However, this does not eliminate the effort of writing specifications because, in the end, someone must reason about the detailed behavior of the hardware. In this paper, we evaluated whether LLMs can be the ones to reason about the hardware behavior and generate, in our case, a low-level specification in SVA.

We found that GPT4—with careful guidance—can do that; it does not merely translate Verilog into SVA but rather seems to capture some of the design intent. Integrated into the AutoSVA framework, GPT4 enables the automatic generation of FTs for RTL modules, whose completeness and correctness depend on the complexity of the design. For small hardware components like the queue we evaluated, it requires very little human intervention to get a complete FT.

We argue that AutoSVA2 has the potential to expand the adoption of FPV—much needed in this era of heterogeneous hardware. Moreover, producing FTs exclusively from RTL could pave the way for safer LLM-assisted RTL design approaches [14]. We also believe that with a curated dataset of SVA properties and their RTL, there is potential for fine-tuning LLMs to be more accurate or cost-effective (with smaller models). Perhaps AutoSVA2 can be a starting point for generating such a dataset from open-source RTL; ***AutoSVA2 can assist you*** in generating FTs for existing RTL modules and developing new ones from scratch. To conclude, we want to encourage the community to keep advancing and streamlining RTL design and verification methodologies that leverage the power of FPV. Our ***open-source artifacts*** [2] include (in addition to our experiments' outputs): our rule set to guide GPT4 at generating SVA (`SVA_GEN.v`) and AutoSVA annotations (`AUTOSVA_GEN.v`); our template to prompt GPT4 to generate RTL (`RTL_GEN.v`); and the script that puts everything together (`autosva2.py`).

References

[1] "AutoSVA," github.com/PrincetonUniversity/AutoSVA.
[2] Anonymous, "AutoSVA2," 2023, https://github.com/gpt4rtl/AutoSVA2.
[3] Cadence Design Systems Inc., "Jaspergold apps user's guide," 2015.
[4] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," *arXiv preprint arXiv:2303.04864*, 2023.
[5] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip verification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.
[6] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2012 Std., 2013.
[7] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "LLM-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.
[8] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *2017 50th Annual IEEE/ACM MICRO*, 2017, pp. 463–476.
[9] OpenAI, "GPT4," https://openai.com/gpt-4.
[10] OpenHW Group, "CVA6," https://github.com/openhwgroup/cva6.

[11] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "AutoSVA: Democratizing Formal Verification of RTL Module Interactions," in *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 535–540.

[12] M. Orenes-Vera, H. Yun, N. Wistoff, G. Heiser, L. Benini, D. Wentzlaff, and M. Martonosi, "AutoCC: Automatic discovery of covert channels in time-shared hardware," in *56th IEEE/ACM MICRO*, 2023.

[13] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015.

[14] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *DATE*. IEEE, 2023, pp. 1–6.

[15] C. Wolf, "SymbiYosys," https://github.com/YosysHQ/SymbiYosys.