Can LLMs Perform Synthesis?

Derek Egolf , Yuhao Zhou , and Stavros Tripakis
Northeastern University, Boston, MA, USA
{egolf.d,zhou.yuhao,stavros}@northeastern.edu

***Abstract*—How do LLMs compare with symbolic tools on program synthesis tasks? We investigate this question on several synthesis domains: LTL reactive synthesis, syntax-guided synthesis, distributed protocol synthesis, and recursive function synthesis. For each domain, we choose a state-of-the-art symbolic tool and compare it to an open-source, 32 billion parameter version of the QWEN LLM and the proprietary, frontier LLM GPT-5. We couple QWEN with a symbolic verifier and run it repeatedly until it either produces a solution that passes the verifier, or until there is a timeout, for each benchmark. We run GPT-5 once per benchmark and verify the generated output. In all domains, the symbolic tools solve more benchmarks than QWEN and either outperform or are about on par with GPT-5. In terms of execution time, the symbolic tools outperform QWEN and GPT-5 in all domains, despite the fact that the LLMs are run on significantly more powerful hardware.**

I. INTRODUCTION

The goal of program synthesis has been to generate, as automatically as possible, systems that are *correct by construction*, that is, that are guaranteed to satisfy a given correctness specification [18], [29]. Since the publication of Church's problem in the 1950s [18], synthesis has come a long way, from the early approaches of deductive program synthesis [53] and reactive synthesis [62], [29], to more recent approaches such as sketching [70], [69], syntax-guided synthesis [5], and others [31].

On the other hand, the field of programming itself is currently undergoing rapid changes; LLMs are now routinely used to generate code for common programming tasks [57], [17], [78].

These developments lead naturally to the question: *can LLMs do synthesis?* Specifically, we want to evaluate how good LLMs are at synthesizing programs from formal specifications, compared to symbolic tools. Therefore, we rephrase our research question as follows: *how do LLMs compare with state-of-the-art symbolic synthesis tools?*

Choice of synthesis domains: As there are many kinds of synthesis problems, the answer to the above question generally depends on the particular synthesis domain that is chosen. In this paper, we examine the following synthesis domains:

- LTL reactive synthesis [62], [29]: the problem is to synthesize a finite state machine (FSM) which implements (*realizes*) a given specification in Linear Temporal Logic (LTL) [61].
- Syntax-guided synthesis (SyGuS) [5]: the problem is to synthesize, given a specification and a grammar, a program that can be generated by the grammar and that satisfies the specification.
- Distributed protocol synthesis by sketching [23], [24]: given a sketch of a distributed protocol in TLA$^+$ [44], the problem is to complete the sketch such that the resulting protocol satisfies a given set of TLA$^+$ properties.
- Recursive program synthesis by sketching [25]: given a sketch of a recursive function in the LISP-like ACL2s programming language [21], and a specification in first-order logic, the problem is to complete the sketch such that the resulting program satisfies the specification.

Choice and use of LLMs: Across all domains, we compare the corresponding symbolic tool with two LLMs: QWEN-2.5-CODER-32B-INSTRUCT [35] (hereafter referred to as QWEN) and GPT-5 [68]. QWEN is a 32 billion parameter model. To our knowledge, it is the state-of-the-art open-source LLM for code generation that fits on an NVIDIA H200 GPU (the hardware that we use to run it). GPT-5 is a proprietary model, and to our knowledge OpenAI has not publicly disclosed the number of parameters it uses. GPT-3, a model released in 2020, had 175 billion parameters [11], so we can reasonably assume that GPT-5 has significantly more than that.

We emphasize that, in all cases, we use these LLMs "out of the box" meaning that we do not train the LLMs ourselves, we do not fine-tune them, we do not use RAG or similar techniques, and so on. In the spirit of our original question *can LLMs do synthesis?*, we want to evaluate to what extent LLMs are able to do synthesis *on their own*, as opposed to how LLMs might be integrated into more sophisticated "hybrid", "neurosymbolic", or similar techniques. We do believe that such techniques are important, but they are beyond the scope of our comparison here. (We review related neurosymbolic techniques in our related work section §VII.)

At the same time, we take a pragmatic, user-oriented perspective. How might a user use an LLM for code synthesis? LLM outputs come with no guarantees, so a user will have to check the output to ensure that it satisfies the problem requirements. For that reason, we couple LLMs with a symbolic *verifier* which checks the LLM output, and potentially has the LLM retry in case of failure. In the spirit of fairness, we try to allocate similar resources to the toolchains under comparison, as much as possible.

Section II that follows presents in more detail the way we use LLMs, discusses fairness considerations and resource allocation, and describes other common elements across all

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_70

domains. Sections III-VI present the experimental results for each domain. Section VII discusses related work. Section VIII concludes the paper.

II. Methodology and Common Elements Across Domains

Fair comparison: A fair comparison of two tools ideally requires running both tools on the same computing hardware and allocating the same time and memory resources to both. This is impossible to achieve in our case, because, unlike all the symbolic synthesis tools that we consider, neither QWEN nor GPT-5 can be effectively run on a CPU. As we still want our comparison to be as fair as possible, our philosophy is to provide as equal as possible computational resources to the compared toolchains. By *computational resources* we mean the product of *hardware computing power* and *time allocated to perform the computation* (which we control by means of *timeouts*). Ideally, we would like this product to be equal so that, for example, if tool A is run on a hardware twice as fast as that of tool B, then the timeout for tool B should be two times that of tool A. In practice, however, this calculation is impossible to achieve, because the computing power of hardware is difficult to measure/compare across hardware architectures. Moreover, to our knowledge, OpenAI does not even disclose the exact hardware used to run GPT-5.

We therefore take a pragmatic approach, and proceed as follows:

- We run all symbolic tools on CPUs, with a 10-minute timeout for each benchmark (15 mins for the recursive program synthesis domain in §VI).
- We use a single NVIDIA H200 GPU to run QWEN. We couple QWEN with a verifier (run on a CPU) which checks each candidate that QWEN generates. For each benchmark, the whole process is iterated until a correct solution is found, or a 10-minute time limit is reached (15 min for §VI).
- We run GPT-5 using the OpenAI API. In all domains, we set an output token budget of 16,384 and set the reasoning parameter to "medium". For each benchmark, we run GPT-5 once, and check any generated solution with a verifier (run on a CPU), allocating a total of 10 minutes both for GPT-5 generation and verification (15 min for §VI).

The rationale is to give the same timeout to each toolchain, even though the toolchains are executed on different hardware. To compensate for the fact that GPT-5 is itself very powerful and also run on very powerful hardware, we only query GPT-5 once per benchmark, and do not iterate like we do with QWEN. This is still a somewhat unfair comparison between symbolic tools and QWEN, as the latter is run on much more powerful hardware. We accept this until a better comparison methodology is found.

LLM toolchains: Pictorially, our LLM toolchains across all domains look like the one in Figure 1. Note that the QWEN toolchain has the *reprompt* loop, but the GPT-5 toolchain does not (GPT-5 is only queried once per benchmark). We handle timeouts as follows. In the GPT-5 toolchain, we allocate a 10-min time limit (15 min for §VI) to the entire toolchain, per benchmark. This means, for example, that if GPT-5 takes 3 mins to generate a candidate solution, then the verifier has 7 mins to check that candidate. A solution which is both generated and verified correct within 10 mins is counted as a correct solution. In the QWEN toolchain, we allocate 10 mins per benchmark (15 min for §VI), and let the loop iterate as many times as possible until the 10 mins are exhausted, or a correct solution is found. Again, we count a solution as correct if it is both generated and verified within the time limit.

We call this method *iterate-LLM-until-solution-or-timeout* (ILST). We remark that, for a given benchmark, the prompt remains the same for each iteration of the loop. That is, we do not provide any verification feedback to the LLM: no counterexamples and no other indication that the previously generated solutions were incorrect. This would not only require adding the previously generated (incorrect) solutions to the prompt, but it would also go against our premise to test whether LLMs can do synthesis *on their own*, and not as part of a neurosymbolic loop. It therefore often happens that the LLM generates the same solution many times during the loop, despite using temperature and other parameters which make the generation more flexible and less deterministic (for all QWEN toolchains we use temperature=0.8, top_p=0.95, and top_k=50). In some domains, we cache the generated solutions within the ILST loop, and if the same solution is generated twice, we skip the verification step since we already know that this solution is incorrect. Using a higher temperature parameter could increase diversity, but it could also increase syntax or semantic failures. A systematic and thorough evaluation of that tradeoff and in general of the effect of temperature (and possibly also the many other LLM parameters) on performance, would require a separate study and is beyond the scope of the current paper.

We remark that we also considered the option of asking the LLM to generate a bundle of, say $k > 1$, different solutions, and then verifying all of them. We opt instead to use ILST for the following reasons. First, in the k-bundle approach, it is unclear how to handle verification time budgets in a meaningful way. For example, what happens if the LLM generates $k = 2$ solutions and the first one is model-checked to be incorrect while the second one is model-checked correct but the model-checker exceeds its timeout? In contrast, in ILST, the time spent generating and verifying is automatically balanced appropriately. Second, ILST more closely resembles how a human would use the LLM: generate a solution, check it, and try again if necessary. Finally, many symbolic synthesis tools are *enumerative* in nature, meaning that they enumerate a candidate solution, check it, and if incorrect, try again. This similarity makes ILST a natural fit for comparison against symbolic tools.

Artifact: Our artifact and final experimental results are available in a permanent repository [27].

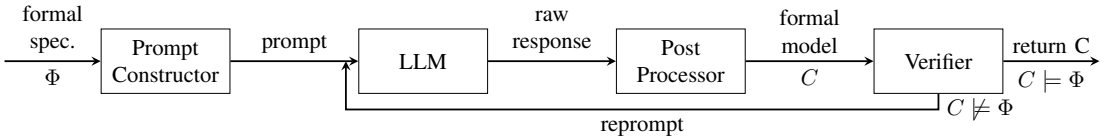

Fig. 1: LLM-based synthesis methodology: the QWEN toolchain has the *reprompt* loop, but the GPT-5 toolchain does not (GPT-5 is only queried once per benchmark).

III. LTL Reactive Synthesis

The problem is to synthesize an FSM which implements (*realizes*) a given LTL specification [62], [29]. We adopt the problem format from the LTL Synthesis Track of SYNTCOMP [37] – https://www.syntcomp.org/. SYNTCOMP uses the Temporal Logic Synthesis Format (TLSF) [36] for specifications. A TLSF file declares the input and output variables of the FSM to be synthesized, and the LTL formulas over these variables that the FSM must implement.

A. Experimental Setup

Symbolic tool: We compare LLMs with the state-of-the-art synthesis tool `ltlsynt` [65], which solved the most benchmarks in the LTL Synthesis Track of SYNTCOMP 2025. `ltlsynt` produces solutions in the AIGER format [10]. We allocate a 10-minute timeout to `ltlsynt` for synthesis. We adhere to SYNTCOMP's verification methodology and employ its verification pipeline to check all results of `ltlsynt` for correctness. First, we use `SyFCo` [36] to convert the TLSF input into SMV/`nuXmv` format, which is subsequently compiled into an AIGER monitor. This monitor is then combined with the candidate solution, and the combined AIGER file is checked by the `nuXmv` model checker [13]. We allocate a 10-minute timeout for this entire verification pipeline. Such post-synthesis verification is unnecessary in principle for a symbolic tool like `ltlsynt` which guarantees correctness by construction. However, we still perform it in order to ensure correctness and maintain consistency with the LLM experiments.

LLM Toolchains: As the choice of output format can impact LLM performance, for this domain, we ask each LLM to synthesize the FSM either in AIGER [10] (the output format of SYNTCOMP) or in SMV (the input format of `nuXmv`). QWEN$^{\text{AIG}}$ and GPT-5$^{\text{AIG}}$ (respectively, QWEN$^{\text{SMV}}$ and GPT-5$^{\text{SMV}}$) denote the LLM toolchains that generate AIGER (respectively, SMV) outputs. The AIGER toolchains use different prompts and verifiers than the SMV ones.

For QWEN toolchains we use an ILST loop (c.f. §II) with a cumulative 10-minute timeout, while for GPT-5 toolchains we use a single-pass generation with a combined 10-minute timeout for generation and verification. For QWEN toolchains we impose a token limit of 16,384 tokens per inference, but we allow the model to generate as many candidate solutions as needed within the time constraint. Each individual QWEN generation is capped at the relatively large limit of 16,384 tokens, which serves only to prevent excessively long solutions. For GPT-5 toolchains we enforce a combined 10-minute timeout for generation and verification, subject to the token budget described in §II.

For verification, the AIGER toolchains use the same verification procedure as `ltlsynt`'s post-synthesis verification described above. Any syntactic errors, such as malformed AIGER files or inconsistencies between the solution's input/output symbols and the TLSF specification, are automatically detected during this verification procedure. In the SMV toolchains, we convert the TLSF specifications to SMV format using `SyFCo`, combine them with the LLM-generated SMV module, and model-check the result with `nuXmv -ctt`. The `-ctt` flag ensures that the transition relation is total (this, for instance, prevents the LLM from "cheating" by generating solutions that trivially satisfy the LTL formulas by deadlocking). We remark that FSMs written in AIGER are deterministic by construction, but LLMs that output SMV could theoretically generate nondeterministic FSMs (but after carefully crafting the prompt, this ends up never happening in our experiments). We therefore check that the results of the SMV LLM toolchains are deterministic. We perform this check only post-synthesis, and not during the ILST loop, because checking determinism is computationally expensive (see the appendix of the full version of this paper [26]). Inside the loop, we rely solely on `nuXmv -ctt`.

Prompts: For the AIGER LLM toolchains, the prompt instructs the LLM to synthesize an implementation of the input TLSF specification in the ASCII AIGER format. The prompt consists of a brief description of the ASCII AIGER format followed by illustrative examples that demonstrate expected LLM's output in AIGER. The complete prompt is provided in the appendix of the full version [26].

The prompt for the SMV toolchains is designed to ensure that the LLM generates a valid, deterministic FSM. To achieve this, we impose strict structural constraints. First, a rigid mapping between TLSF concepts and SMV sections: (1) all TLSF inputs must be declared in an `IVAR` section to prevent assignments to input variables (which can only be controlled by the environment); (2) internal state variables must be declared in the `VAR` section; (3) all variables must be of type `boolean`; (4) TLSF outputs must be defined in the `DEFINE` section. Second, to avoid nondeterministic solutions, we request that the FSM is defined using only an `ASSIGN` block with explicit `init()` and `next()` assignments for every state variable, and we forbid the use of `INIT` and `TRANS` sections. We also provide explicit instructions on boolean operator syntax to minimize syntax errors during generation. Finally, as none of the benchmarks in our suite

require a Moore target FSM (where outputs depend exclusively on state), we do not add any relevant instructions in our prompt. The complete prompt is provided in the appendix of the full version [26].

Benchmarks: We use benchmarks from the LTL Synthesis Track of SYNTCOMP 2025. There are 1586 benchmarks in that suite. Some of these benchmarks are labeled as realizable, unrealizable, or unknown. For our experiments, we select all benchmarks marked as realizable, a total of 433 benchmarks. We only select realizable benchmarks in order to avoid confusing the LLMs.

Hardware and Parameters: We ran `ltlsynt` on a machine with 4 CPU cores and 64GB RAM. The QWEN toolchains ran with the same hardware specification plus an NVIDIA H200 GPU. We ran GPT-5 via the official OpenAI API. We use the parameters described in §II for the QWEN and GPT-5 toolchains.

B. Experimental Results

The results are presented in Table I. The *Solved* column reports the number of correct solutions obtained within the 10-minute timeout. `ltlsynt` generated solutions for 396 out of the 433 benchmarks. We were able to verify (post-synthesis) 354 of those solutions; for the remaining 42, `nuXmv` timed out after 10 minutes (this 10-minute limit applies only to post-synthesis verification and is not counted towards the 10-minute timeout allocated to `ltlsynt`). In the $433-396=37$ benchmarks that `ltlsynt` did not solve, it timed out 34 times (generating no solution) and crashed 3 times (e.g., due to memory allocation failure).

`ltlsynt` solved 143 benchmarks that no LLM toolchain managed to solve. GPT-5$^{\text{AIG}}$ solved 2 benchmarks that `ltlsynt` and the other LLM toolchains failed to solve. Similarly, GPT-5$^{\text{SMV}}$ solved 3 benchmarks that no other method could solve. We also observed that `ltlsynt` solved every benchmark that the QWEN toolchains successfully solved, while QWEN$^{\text{AIG}}$ and QWEN$^{\text{SMV}}$ solved 1 and 6 benchmarks that GPT-5$^{\text{AIG}}$ and GPT-5$^{\text{SMV}}$ failed to solve, respectively. In total, 401 out of 433 benchmarks were solved by at least one method.

The *Success Time* columns present execution time statistics (minimum, maximum, and average execution time in seconds) only for the successful cases, where the toolchains generate a correct solution. For `ltlsynt`, we count all 396 generated outputs as correct solutions for the purpose of execution time statistics. As shown, `ltlsynt` significantly outperforms the LLM toolchains in the execution time metric.

The *Success Iteration* columns present the minimum, maximum, average, and median number of iterations required for the QWEN ILST loops to find a solution (successful cases only). QWEN$^{\text{SMV}}$ sometimes generates a correct solution on the first try, while in other cases it requires up to 135 attempts (recall that no feedback is provided to the LLM). The *Fail Iteration* columns present the same statistics for the failed cases. The minimum of 1 for QWEN$^{\text{SMV}}$ occurs because, in some cases, `nuXmv` timed out while verifying the very first candidate produced. Iteration statistics are omitted for GPT-5 as it is queried only once per benchmark and therefore not iterated. Iteration statistics are also omitted for `ltlsynt` which is a game-based synthesis pipeline, rather than an explicit iterative refinement loop.

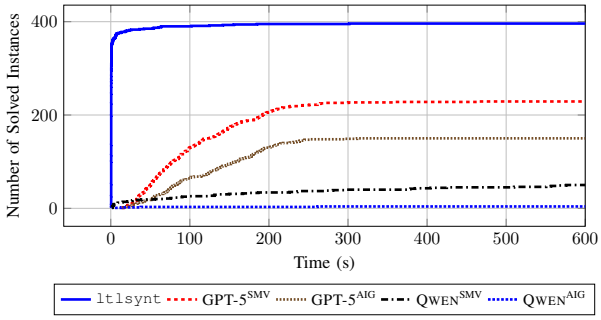

Fig. 2: If each benchmark is allowed the time budget shown in the x-axis, the y-axis shows how many benchmarks can be solved by the corresponding tool, for the LTL synthesis domain.

Because the ILST loop iterates until a correct solution is found or the time budget is exhausted, QWEN can produce the same candidate more than once for a given benchmark. Across the 433 benchmarks, QWEN$^{\text{AIG}}$ produced 17,974 candidates, of which 16,624 were distinct; the remaining 1,350 (7.5%) repeated an earlier candidate for the same benchmark. For QWEN$^{\text{SMV}}$, 15,403 of the 17,338 candidates were distinct, and the remaining 1,935 (11.2%) were repeats.

Cactus Plots: In addition to the timing statistics in Table I, we provide a cactus plot in Figure 2. The point $(t,\ c)$ on each curve means that the corresponding method can solve c benchmarks if each benchmark is given a time budget of t seconds.

Determinism Checking of LLM SMV Solutions: As discussed above, we verify all LLM SMV solutions post-synthesis, to ensure that they represent deterministic FSMs. We checked the determinism of all solved LLM-generated SMV benchmarks of Table I automatically, subject to a 10-minute timeout per benchmark. We used a standard *self-composition* technique which consists in instantiating two copies of the FSM, feeding the same inputs to both, and checking that in all reachable global states, the two FSMs have identical local states (see the appendix of the full version [26] for details). All 50 QWEN$^{\text{SMV}}$ solutions were verified as deterministic within the timeout. Out of the 229 GPT-5$^{\text{SMV}}$ solutions, 225 were verified automatically as deterministic. In the remaining 4 cases the automatic determinism check timed out, but we confirmed the determinism of all these 4 solutions through manual inspection.

LLM Response Failures: QWEN$^{\text{SMV}}$ solved 50 benchmarks and reached the 10-min timeout in the remaining $433-50=383$ cases. In 276 of these 383 cases, a final candidate was generated by the LLM, but was not model-checked (due to the timeout having been reached). Since 276 is a large percentage of 383, we used `nuXmv` to check whether any of those "last-

Method	# Solved	Success Time			Success Iterations				Fail Iterations			
	within TO	Min	Max	Mean	Min	Max	Mean	Med	Min	Max	Mean	Med
`ltlsynt`	354+42	0.5	301.6	4.6	–	–	–	–	–	–	–	–
QWEN$^{\text{AIG}}$	4	5.4	261.5	79.6	5	136	39.2	8	2	522	41.5	7
QWEN$^{\text{SMV}}$	50	1.5	575.9	164.9	1	135	23.9	9	1	202	42.1	33
GPT-5$^{\text{AIG}}$	150	14.9	311.4	124.0	–	–	–	–	–	–	–	–
GPT-5$^{\text{SMV}}$	229	13.8	465.3	107.0	–	–	–	–	–	–	–	–

TABLE I: Summary statistics on 433 LTL reactive synthesis benchmarks.

minute" solutions happened to be correct. All 276 turned out to be wrong: 134 were syntactically incorrect (i.e., files that `nuXmv` could not parse) and 142 were semantically incorrect (i.e., violating the LTL formulas).

GPT-5$^{\text{AIG}}$ ran out of its token budget in 90 cases, generating no solution. In all remaining $433-90 = 343$ cases, it generated a solution within the 10-min timeout. These 343 candidates were then verified by `nuXmv` (with the timeout set to the remaining time within the total 10-minute limit). Of these 343 candidates, 150 passed the model checker (correct solutions); 145 failed due to syntax errors; 48 failed due to semantic errors; and no cases timed out during model checking.

GPT-5$^{\text{SMV}}$ exhausted its token budget in 72 cases. In all remaining $433-72 = 361$ cases, it generated a solution within the 10-minute time limit. These 361 candidates were then verified by `nuXmv` (with the timeout set to the remaining time within the total 10-minute limit). Of these 361 candidates, 229 passed the model checker (correct solutions); 51 failed due to syntax errors; 40 failed due to semantic errors; and 41 timed out during model checking. We reran `nuXmv` on the latter 41 cases with a fresh 10-minute timeout, but all still timed out during model checking (recall that for GPT-5 we use a total 10-minute timeout for both generation and verification, so the original `nuXmv` runs generally had less than 10 minutes available).

GPT-5*: Tokens and Monetary Cost:* For the GPT-5$^{\text{AIG}}$ experiment, the average input length was 1,831 tokens per benchmark, and the average output length was 10,004 tokens. The total cost for all 433 queries was 43.97 USD. For the GPT-5$^{\text{SMV}}$ experiment, the average input length was 1,737 tokens per benchmark, and the average output length was 8,930 tokens. The total cost for all 433 queries was 39.58 USD.

IV. Syntax-Guided Synthesis

The SyGuS problem is to synthesize, given a specification and a grammar, a program that (1) can be generated by the grammar and (2) satisfies the specification [5]. We adopt the SyGuS Input Format (SyGuS-IF) 2.1 standard [59] for expressing SyGuS problems. A SyGuS-IF problem description consists of three main components: a background theory (e.g., linear integer arithmetic), one or more function signatures to synthesize along with optional grammar constraints, and a set of semantic constraints that the synthesized functions must satisfy.

A. Experimental Setup

Symbolic tool: We compare LLMs with the state-of-the-art tool `cvc5` [7]. Its precursor `cvc4` [8] was the winner of 4/5 tracks of the last SyGuS Competition [6], held in 2019. Given a SyGuS-IF 2.1 specification, `cvc5` attempts to synthesize function implementations in SMT-LIB format [9]. We use `cvc5` in its default setting, without supplying any additional command line arguments to it. For this synthesis task, we allocate a 10-minute timeout to `cvc5`.

We verify the solutions produced by `cvc5` for both grammatical and semantic correctness. (In principle this is unnecessary, as `cvc5` guarantees correctness by construction. However, we still perform this post-synthesis verification to ensure correctness and maintain consistency with the LLM experiments.) First, a custom grammar checker ensures that the generated solutions conform to any syntactic constraints defined in the original SyGuS-IF specification (if no grammar is specified, this check trivially passes). Second, to verify semantic correctness, we use a custom tool to convert the SyGuS-IF specification into an SMT query; we then combine this query with the synthesized function implementations and verify it using the Z3 solver [20]. We rely on custom tools for these verification steps because the standard utilities from SyGuS-COMP 2019 can only process SyGuS-IF 1.0 files, whereas `cvc5` and our benchmarks adhere to SyGuS-IF 2.1. The timeout for these post-synthesis verification tasks is set to 10 minutes per benchmark.

LLM toolchains: We evaluate a QWEN ILST loop and a GPT-5 toolchain, as described in §II. Both are instructed to generate SMT-LIB output. In both cases we apply the same verification procedure as for `cvc5`: solutions are checked for both grammar conformance and semantic correctness. A solution is *correct* if and only if it passes both checks. In the QWEN toolchain, the ILST loop iterates until a correct solution is found, subject to a cumulative timeout of 10 minutes per benchmark. Each individual QWEN generation is capped at 4,096 tokens, a relatively large limit that serves only to prevent excessively long inferences. No inference reached this cap in our experiments. In the GPT-5 case we generate a single candidate solution per benchmark, subject to the token budget described in §II. This candidate is then verified for both semantic correctness and grammar conformance. We use a total 10-min timeout for the entire generation and verification process.

Prompts: Both the QWEN and GPT-5 toolchains are provided with an identical prompt. The prompt instructs the model to generate SMT-LIB function implementations that

satisfy the grammar and semantic constraints of the SyGuS-IF input. We strictly constrain the output format: the LLM must produce only self-contained `(define-fun ...)` expressions that match the signature of the target `synth-fun`, without helper functions or natural language explanations. To prevent common syntax errors, the prompt explicitly lists valid SMT-LIB primitives. For instance, it enforces the use of `ite` for conditionals (rejecting `if`), and requires correctly typed bitvector literals (e.g., `(_ bv10 32)`). The full prompt is provided in the appendix of the full version [26].

Benchmarks: We select our benchmarks from the `test/regress/cli` directory in the `cvc5` repository [72], which contains more recent problems than the SyGuS Competition (SyGuS-COMP) [6], last held in 2019. We extract from the above directory all realizable SyGuS-IF specifications, excluding those requiring optional SyGuS-IF features (c.f. Appendix C in [59]) or experimental `cvc5` features. This extraction yields 148 benchmarks in total.

Hardware and Parameters: We ran `cvc5` on a machine with 4 CPU cores and 64GB RAM. The QWEN toolchain ran with the same hardware specification plus an NVIDIA H200 GPU. We ran GPT-5 via official OpenAI API. We use the parameters described in §II for the QWEN and GPT-5 toolchains.

B. Experimental Results

The results are summarized in Table II. The *Solved* column reports the number of benchmarks for which a correct solution was found within the 10-minute timeout. `cvc5` successfully solved 137 out of 148 benchmarks. All 137 solutions were confirmed as correct by our post-synthesis verification. The meaning of the other columns is the same as in the corresponding columns of Table I. As for other domains, we omit iteration statistics for GPT-5 since it is always queried once per benchmark. We also omit iteration statistics for `cvc5` because its public statistics interface does not provide an iteration count.

`cvc5` solved 4 benchmarks that no LLM toolchain could solve. GPT-5 solved 10 benchmarks that `cvc5` failed to solve. We also observed that GPT-5 solves every benchmark that QWEN solved, while QWEN solved 6 benchmarks that `cvc5` failed to solve. In total, 147 out of 148 benchmarks were solved by at least one method.

Because the ILST loop iterates until a correct solution is found or the time budget is exhausted, QWEN can produce the same candidate more than once for a given benchmark. Across the 148 benchmarks, QWEN produced 24,059 candidates, of which only 2,288 were distinct; the remaining 21,771 (90.5%) repeated an earlier candidate for the same benchmark.

Cactus Plots: Figure 3 shows cactus plots for this domain, following the same convention as for Figure 2.

LLM Response Failures: The QWEN toolchain solved 96 benchmarks and reached the 10-minute timeout in the remaining $148 - 96 = 52$ benchmarks. In all 52 failed cases, the LLM generated a final candidate that was not verified due to the timeout. We used the verifier to check if these "last-minute" solutions were correct (note that they are not counted as "solved" in Table II, even if verified to be correct). Of these 52 candidates, 2 were actually correct, 46 were semantically incorrect (i.e., violating grammar or specification constraints), and 4 were syntactically incorrect (i.e., causing parsing errors in the verifier).

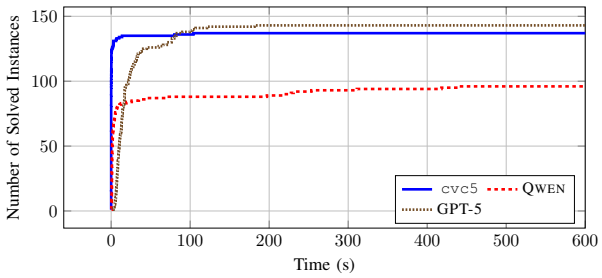

Fig. 3: If each benchmark is allowed the time budget shown in the x-axis, the y-axis shows how many benchmarks can be solved by the corresponding tool, for the SyGuS domain.

The GPT-5 toolchain failed to solve 5 out of 148 benchmarks. Among these 5 failures, 3 were semantically incorrect, and 2 were caused by GPT-5 exceeding the token budget.

GPT-5*: Tokens and Monetary Cost:* For the GPT-5 toolchain, the average input length was 1,340 tokens per benchmark, while the average output length was 1,609 tokens. The total cost for all 148 queries was 2.62 USD.

V. TLA$^+$ Distributed Protocol Synthesis

Given a *sketch* of a distributed protocol in TLA$^+$ [44], the synthesis problem is to *complete* the sketch such that the completed protocol satisfies a given set of properties. A sketch is a TLA$^+$ protocol with holes, along with grammars that define the set of expressions that can be used to fill the holes. If the protocol is *parameterized*, e.g. by the number of nodes participating in the protocol, we assume that the parameters are finitized to specific values, which is standard practice [24].

A. Experimental Setup

Symbolic Tool: Our evaluation includes the state-of-the-art symbolic distributed protocol synthesis tool POLYSEMIST [24]. POLYSEMIST uses an *enumerative, counterexample-guided* synthesis algorithm. Internally, it enumerates candidate protocols, applies the TLC model checker [77], and halts if a candidate protocol satisfies the properties. If not, it continues enumerating candidates. Therefore, POLYSEMIST cannot return a protocol that violates the properties specified by the user [24]. Because POLYSEMIST enumerates protocols from the language of the grammar [24], the synthesized protocol is guaranteed to conform to the grammar.

LLM Toolchains: We instantiate GPT-5- and QWEN-based toolchains for this domain, and we use the TLC model checker [77] as the core verification engine. Because grammar conformance is particularly important in this domain [23],

Method	# Solved	Success Time			Success Iterations				Fail Iterations			
	within TO	Min	Max	Mean	Min	Max	Mean	Med	Min	Max	Mean	Med
cvc5	137	0.01	104.1	1.8	–	–	–	–	–	–	–	–
QWEN	96	0.54	439.6	27.8	1	450	16.3	1	15	921	432.5	466.5
GPT-5	143	2.35	270.1	28.7	–	–	–	–	–	–	–	–

TABLE II: Summary statistics on 148 SyGuS benchmarks.

[24], our QWEN toolchain also uses a custom grammar verifier which checks that the LLM responses conform to the provided grammar, before passing the response to TLC. We use the same custom grammar verifier to check the solutions generated by GPT-5. Additionally, for this domain, our QWEN ILST loop maintains a cache mapping LLM responses to model checking results. If an LLM repeats itself, we skip model checking because we can infer that TLC has already rejected this response.

Prompt: The prompt indicates which holes the LLM should fill, the actions where these holes appear, the grammars for each hole, the properties that the completed protocol must satisfy, and the structure of the protocol in JSON format. We show the prompt template for this domain in the appendix of the full version [26].

Benchmarks: In [24], POLYSEMIST was evaluated on a suite of 171 benchmarks, publicly available at [22]. We use these same benchmarks for our evaluation. These benchmarks cover seven distributed protocols, including two reconfigurable raft protocols. The benchmarks are to fill in missing expressions within one or two actions of the protocol. These expressions are either all pre-conditions for these actions, all post-conditions, or all pre- and post-conditions.

Hardware and Parameters: We run POLYSEMIST on a machine with an AMD Ryzen 7 7840U CPU (3.3 GHz base clock) and 14GB of RAM. The QWEN toolchain ran on a machine with an Intel(R) Xeon(R) Platinum 8276 CPU (2.20GHz), 16GB RAM, plus an NVIDIA H200 GPU. We run GPT-5 using the OpenAI API. For GPT-5, we model check the responses obtained from the API on the same machine used for POLYSEMIST. The TLC model checker was run with 8 workers in all toolchains and all toolchains had access to at least 8 CPU cores. There are no other sources of parallelism. We use the parameters described in §II for the QWEN and GPT-5 toolchains.

B. Experimental Results

The results are shown in Table III. The *Solved* column shows for how many of the 171 benchmarks the corresponding method produces correct solutions, meaning solutions that pass both the model checker and the grammar verifier. As shown in Table III, the symbolic tool POLYSEMIST solves more benchmarks than GPT-5, and QWEN solves fewer. There are benchmarks that are solved by the LLMs but not by POLYSEMIST, and vice versa. Together, GPT-5 and QWEN solve 17 benchmarks that POLYSEMIST does not solve. Of these 17, GPT-5 alone solves 11, QWEN alone solves 1, and there are 5 benchmarks that both LLM toolchains solve but POLYSEMIST does not. There are 16 benchmarks that

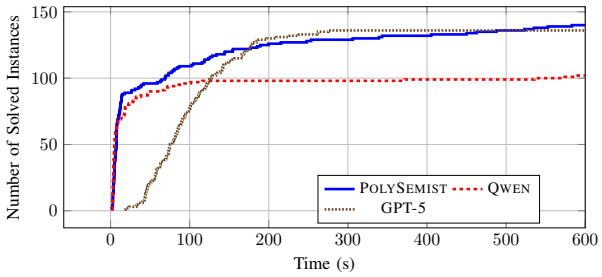

Fig. 4: If each benchmark is allowed the time budget shown in the x-axis, the y-axis shows how many benchmarks can be solved by the corresponding tool, for the TLA+ domain.

POLYSEMIST solves that neither LLM toolchain solves. There are 28 benchmarks that both GPT-5 and POLYSEMIST solve, but QWEN does not. There are 4 benchmarks that both QWEN and POLYSEMIST solve, but GPT-5 does not. There are 92 benchmarks that all three solve and 14 benchmarks that none of the three solve.

The meaning of the remaining columns of Table III is the same as the corresponding columns in Table I. In the case of POLYSEMIST we also report iteration statistics, which are available for this symbolic tool [24]. Iteration statistics are omitted for GPT-5 because it is queried only once per benchmark.

Cactus Plots: In addition to the timing statistics in Table III, Figure 4 shows cactus plots. The point (t, c) on each curve means that the corresponding method can solve c benchmarks if each benchmark is given a time budget of t seconds (it does not mean that the tool takes t total seconds to solve c benchmarks). The overall trend suggests that POLYSEMIST tends to solve the easier benchmarks faster than GPT-5. In the long run, the two tools perform similarly.

LLM Response Failures: GPT-5 failed to solve 35 out of 171 benchmarks. In 33 of these failures the model checker found counterexamples, and in one case GPT-5 exceeded the token limit. In one case, GPT-5 produced a correct solution after more than 10 minutes, which we count as a timeout.

Because QWEN iterates, we received in total 16,824 responses from the LLM. If we take the *sets* of distinct responses for each benchmark and sum their cardinalities, we get 7,326 distinct responses. This means that QWEN repeats itself (incorrectly) quite often. Of the 16,824 responses, 16,722 were failed responses. In 11,628 of these cases, the model checker found counterexamples. In one case, QWEN produced a correct solution after more than 10 minutes, which we count as a timeout. In 5,063 cases, the QWEN response did not conform to the provided grammar (GPT-5 responses always conformed

Method	# Solved	Success Time (s)			Success Iterations				Fail Iterations			
	within TO	Min	Max	Mean	Min	Max	Mean	Med	Min	Max	Mean	Med
PolySemist	140	0.70	583.06	70.94	1	219	20.7	10	7	272	97.3	99
Qwen	102	2.03	587.07	35.03	1	254	10.8	1	76	717	227.9	202
GPT-5	136	17.55	275.35	102.22	–	–	–	–	–	–	–	–

TABLE III: Summary statistics on 171 TLA$^+$ protocol synthesis benchmarks.

to the provided grammar). In 30 cases Qwen did not even produce syntactically valid JSON (GPT-5 always produced syntactically valid JSON).

Relaxing Grammar Constraints: We also evaluated the LLM toolchains with a relaxed verification engine that does not check for grammar conformance but only applies TLC to check for semantic correctness. In the case of Qwen this means that in the ILST loop, only TLC is used as the verifier, and when a semantically valid solution is found the loop exits. Under this relaxed setting, we also omitted the grammar from the prompt for both Qwen and GPT-5. We found that both LLM toolchains solve more benchmarks in this configuration. This result is not surprising, as the LLMs in this case are solving an easier synthesis problem (without the grammar constraint). Under this relaxation, Qwen solves 133 benchmarks (compared to 102 in Table III), while GPT-5 solves 149 benchmarks (compared to 136 in Table III, and compared to the 140 that PolySemist solves).

GPT-5*: Tokens and Monetary Cost:* We use GPT-5-NoGram to refer to the GPT-5 toolchain where the grammar is not checked (for the relaxation discussed above). For GPT-5, the average number of input tokens per benchmark is 2125, and for GPT-5-NoGram it is 1264. The average number of output tokens per benchmark is 5389 for GPT-5, and 6041 for GPT-5-NoGram. We note that on average the synthesized solution in both cases is only about 70 tokens and the majority of output tokens are the so-called "reasoning tokens." GPT-5-NoGram is given fewer input tokens because the grammar is not included in the prompt. We note that GPT-5-NoGram produces more reasoning tokens; removing the grammar causes the model to "think" more. Overall, the total cost to run all GPT-5 experiments reported for this domain is about 20 USD.

VI. ACL2s Recursive Program Synthesis

The recursive program synthesis problem considered in this domain is to complete sketches for one or more recursive functions such that the program composed of the completed functions satisfies a given set of correctness properties. As in §V, a sketch is a program with holes, along with grammars that define the set of expressions that can be used to fill the holes. In this domain, the programs are written in the LISP-like ACL2s programming language [21], while sketches and grammars are expressed in a similar LISP-like format. The correctness properties are first-order logic formulas. We refer the reader to [25] for details.

A. Experimental Setup

Symbolic Tool: We use Cataclyst [25], a symbolic tool that synthesizes recursive programs from formal specifications. Cataclyst employs an enumerative, counterexample-guided synthesis algorithm. Cataclyst internally checks candidate programs using the verification engine discussed below. A consequence is that Cataclyst can only output programs that pass the verification engine. It also cannot output syntactically invalid programs.

Verifier and Correctness Guarantees: Verification of synthesized programs in this domain amounts to first-order logic validity checking, and is therefore undecidable in general. We therefore follow the approach taken in [25], and use as the verification engine of our toolchains the counterexample generator of the ACL2s theorem prover [21], [15]. A consequence of this is that Cataclyst can, in principle, return a program that passes the counterexample generator but is not actually correct. The authors of [25] use the ACL2s theorem prover to perform post-hoc manual verification of all programs synthesized in [25] and found that all synthesized programs were indeed correct. We therefore take for granted that, at least for the benchmarks in [25] (which we also use here), passing counterexample generation is a good proxy for full-blown verification.

LLM Toolchains: As for the prior domains, the LLM toolchains have the same overall structure as in Figure 1. We instantiate two LLM toolchains: one using Qwen and one using GPT-5. Both toolchains are relaxed, in the sense that they check the correctness properties but do not check that proposed programs conform to the provided sketches or grammars. In this domain, the sketches and grammars are primarily useful for guiding Cataclyst's enumerative search. It is not critical that the synthesized programs conform to the grammar, so long as the programs are syntactically valid. In the interest of fairness though, we give the LLM toolchains the sketches and grammars as input. Here, we also cache LLM responses and skip verification if the LLM repeats itself.

Prompts: For this domain, the prompt template gives: (1) the function signatures of the functions under synthesis, (2) the list of primitive functions that can be used in the synthesized functions, (3) the terminals (constants and function arguments) that can be used in the synthesized functions, (4) the list of properties and input-output examples that the synthesized functions must satisfy, (5) auxiliary information about data types and functions that may be useful context for the LLM, and (6) the sketch for each function and the grammar for filling holes in the sketches. The full prompt template that we use is shown in the appendix of the full version [26].

Benchmarks: We use the 42 benchmarks used to evaluate Cataclyst in [25]. These benchmarks include programs for manipulating lists, trees, and other inductive data structures. Some benchmarks involve mutually recursive functions, and

some benchmarks involve properties with existential quantifiers.

Hardware and Parameters: We ran CATACLYST on a machine with an AMD Ryzen 7 7840U CPU (3.3 GHz base clock) and 14GB of RAM. The QWEN hardware configuration is identical to that described in §V for the TLA$^+$ synthesis domain, except that only one CPU core is allocated. The GPT-5 toolchain ran verification on the same machine used for CATACLYST and uses the same LLM parameters as described in §II. Neither the symbolic tool nor the verifier are configured to use parallelism.

B. Experimental Results

Our results are summarized in Table IV. All methods in this domain use a 15 minute timeout rather than the 10 minute timeout used in prior domains. We use a 15-minute timeout to match the timeout used in the evaluation of CATACLYST [25]; successes were achieved in less than 10 minutes when success was achieved, so a 10-minute timeout would only reduce the failure-iteration counts in Table IV. The *Solved* column reports solutions that pass the ACL2s counterexample generator. The meaning of the other columns is the same as in the corresponding tables of prior domains. Iteration statistics are omitted for GPT-5 because it is queried only once per benchmark. As shown in Table IV, QWEN solved 39/42 benchmarks and both GPT-5 and CATACLYST solved 41/42 benchmarks. QWEN failed on the following three benchmarks: "pairs," "split-on," and "suffixb" [25]. GPT-5 failed only on "split-on." CATACLYST failed only on "ternary-tree-eq." Thus both LLM toolchains failed on "split-on," which CATACLYST solved, while only QWEN failed on "suffixb" and "pairs." Both LLM toolchains succeeded on "ternary-tree-eq," which CATACLYST could not solve.

Iterations and Failures: As shown in Table IV, in this domain, when QWEN solves a benchmark, it tends to do so using just a few iterations (and corresponding LLM queries). On the other hand, when CATACLYST solves a benchmark, it often enumerates hundreds of candidate solutions. When QWEN does not solve a benchmark, it does so after proposing hundreds of incorrect solutions: 179, 414, and 516 proposals for the three unsolved benchmarks. In the single case where CATACLYST fails, it does so after enumerating 4795 candidate solutions.

We count also the number of *distinct* solutions proposed by QWEN in the three failed benchmarks: 91/179, 20/414, 8/516. I.e., in one of the failed tasks QWEN was asked 179 times for a solution, and in all but 91 cases it proposed a solution that it had proposed before. In general, we obtained a total of 1159 responses from QWEN across all benchmarks. Only 165 of those responses were distinct, so QWEN repeated itself many times.

Of the 1159 responses we obtained from QWEN, 1120 were failed responses. 501 of these failures were caused because the counterexample generator disproved correctness of the proposed solution. 619 failures were due to syntactically invalid responses from the LLM, e.g., mismatched parentheses. In the one case where GPT-5 failed, it was due to a counterexample being found.

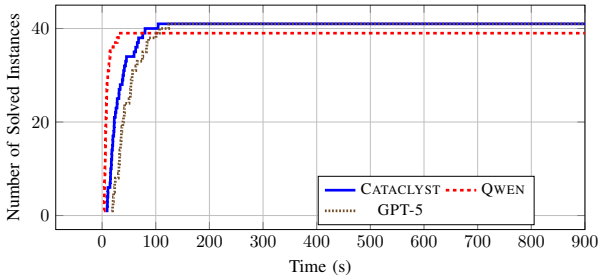

Fig. 5: If each benchmark is allowed the time budget shown in the x-axis, the y-axis shows how many benchmarks can be solved by the corresponding tool, for the ACL2s domain.

GPT-5*: Tokens and Monetary Cost:* For GPT-5, the average number of input tokens per benchmark is 680, the solution is composed of 87 output tokens on average, and GPT-5 outputs an average of 2039 reasoning tokens per benchmark. The total monetary cost for all 42 benchmarks is about 0.93 USD.

VII. RELATED WORK

In the LTL reactive synthesis domain, the works [67], [66], [19] explore neural approaches and evaluate them on SYNTCOMP benchmarks. All these works train or fine-tune *transformer* models for synthesizing AIGER circuits, whereas we evaluate off-the-shelf models without any additional training, and we consider both AIGER and SMV outputs.

In the SyGuS domain, Li et al. [47] propose a hybrid approach that integrates LLM calls into a weighted probabilistic enumerative search. Their evaluation shows that `cvc5` outperforms GPT-3.5 alone, while their hybrid approach outperforms both. Our work compares `cvc5` with the more recent GPT-5; in our experiments, GPT-5 outperforms `cvc5`. We also compare with the 32-billion parameter Qwen-2.5-Coder, whereas [47] only compare with GPT-3.5. Finally, we use a different set of benchmarks than those used in [47].

To our knowledge, [23], [24] are the only works that synthesize protocols in TLA$^+$, and they do so with symbolic tools with no mention of LLMs. The GenAI-accelerated TLA$^+$ challenge [75] called "for submissions showcasing creative and technically impressive work at the intersection of TLA$^+$, formal methods, and AI-assisted development," and announced winners in 2025. The first place winner Specula takes as input program source code (e.g. Rust) and outputs TLA$^+$. The third place winner by Gregory Terzian takes as input TLA$^+$ and produces Rust code. These tools solve TLA$^+$ code generation problems using LLMs, but these problems are different from the protocol synthesis problem we consider in this paper.

CVC [7], [8] can synthesize non-recursive programs, but it cannot synthesize recursive programs [1]. Likewise, none of [30], [43], [3] can synthesize recursive programs. [52], [4], [32], [58], [28] synthesize recursive programs from input-output examples. [55], [54], [63], [42], [25], [33] synthesize

Method	# Solved	Success Time (s)			Success Iterations				Fail Iterations			
	within TO	Min	Max	Mean	Min	Max	Mean	Med	Min	Max	Mean	Med
CATACLYST	41	6.47	104.41	31.80	4	2552	495.6	173	4795	4795	4795	4795
QWEN	39	3.23	30.42	9.03	1	6	1.3	1	179	516	369.7	414
GPT-5	41	17.33	124.16	48.33	–	–	–	–	–	–	–	–

TABLE IV: Summary statistics on 42 ACL2s recursive program synthesis benchmarks.

recursive programs from properties. None of these works compare LLMs with symbolic tools.

There are other synthesis domains that fall outside the scope of this paper. LLMs are used to generate loop invariants in [14], [40], class invariants in [71], spreadsheet formulas in [39], and to perform C program repair in [74]. These works address different synthesis problems.

Several of the works cited above can be considered "hybrid" or "neurosymbolic" in the sense that they combine neural networks and symbolic tools (including verifiers, but in more sophisticated ways than our *iterate-LLM-until-solution-or-timeout* approach). Neurosymbolic synthesis [16] is an active research direction but beyond the scope of our current evaluation.

Several works propose benchmarks for LLM code generation in various domains, including general programming tasks [17], [78], [57], [64], [48], [34], as well as code generation tasks more closely related to this paper. The works in [76], [73], [12], [50] propose benchmarks for verification-oriented languages such as Lean [56], Dafny [46], or Verus [45]. LLM code generation benchmarks for Solidity smart contracts are proposed in [60], and for the domain of hardware in [41], [2], [38], [49], [51]. Several of these works evaluate LLM-generated code with respect to correctness or efficiency, and several employ verification tools for checking correctness. However, the focus of these works is not the comparison of LLMs with symbolic tools, and their domains are different from the ones examined in our work.

VIII. CONCLUSIONS

We compared symbolic synthesis tools with LLM toolchains, across several synthesis domains. Although the results vary by domain, the following observations can be made (c.f. Tables I, II, III, IV and the corresponding sections):
(1) The symbolic tools solve more benchmarks than the QWEN toolchains, in all domains. The difference is dramatic in the LTL synthesis domain.
(2) `ltlsynt` solves significantly more benchmarks than the GPT-5 toolchains in the LTL synthesis domain. GPT-5 solves almost the same number of benchmarks as the corresponding symbolic tool in the other three domains.
(3) There are generally benchmarks that are solved by the LLM but not by the symbolic tool, and vice versa. From a practical perspective, this suggests using a portfolio approach which runs both symbolic and LLM-based toolchains in parallel.
(4) The symbolic tools are faster than the GPT-5 toolchains, in all domains, despite the fact that GPT-5 runs on more powerful (and proprietary) hardware. In the LTL synthesis and SyGuS domains, the difference is two and one orders of magnitude, respectively.
(5) Even though this may not be immediately apparent in Tables I, II, III, IV, the symbolic tools are also faster than the QWEN toolchains, in all domains, and despite the fact that QWEN runs on more powerful hardware. To see this, note first that the tables show timing statistics about *successful* runs only. To get the full picture, the complete execution times must be inferred. For instance, consider Table III. QWEN solves 102 benchmarks with an average of 35.03 secs per benchmark. But QWEN also times out on 69 benchmarks, each after 600 secs. In total, QWEN executed for $35.03 \cdot 102 + 600 \cdot 69 = 44973$ secs and managed to solve 102 benchmarks in that time. In contrast, POLYSEMIST executed in total for $70.94 \cdot 140 + 600 \cdot 31 = 28532$ secs and managed to solve 140 benchmarks in that time. The same calculation can be done for Table IV. Using the 900-second timeout for benchmarks that were not solved, QWEN runs for 3052 seconds, which is more than CATACLYST's 2204 seconds.
(6) QWEN toolchains often repeat the same incorrect proposal many times. An advantage of symbolic tools that use an enumerative synthesis algorithm is that they can be designed so that they never repeat the same candidate. Devising ways to avoid repetitions when using LLMs is part of future work (see also below).
(7) Except for the LTL domain, GPT-5 was quite good at meeting the syntactic expectations that we set for it (well-formed code, grammar/sketch conformance). QWEN struggled with meeting these expectations. Even in the recursive program synthesis domain, where it ultimately solved many benchmarks, more than half of its failures were due to syntactic issues. Symbolic tools are designed to always meet syntactic expectations, which is an advantage.

Future Work: As explained in Sections I and II, the goal of this paper is to test whether LLMs can do synthesis on their own, possibly with several trials (in the case of QWEN), but without feedback. Investigating approaches where, for instance, failed verification results are given as feedback to the LLM, is part of future work. We remark that there are many ways to provide such feedback to the LLM. The simplest approach is to provide a list of erroneous solutions, along with instructions not to output those solutions. A more sophisticated approach is to annotate each bad solution with a counterexample input or trace, which might allow the LLM to infer *why* a solution is bad. Both approaches would result in longer and longer prompts as the number of erroneous attempts grows, which might in turn result in longer inference times or even in exhausting the context budget. Devising methods to address such problems is part of future work.

Prompting approaches bias the LLM towards generating new responses, but they do not offer any guarantees. In future work, we plan to explore a neurosymbolic approach where the LLM is used as a source of statistical bias for an otherwise symbolic synthesis algorithm. Unlike prompting approaches, this approach could make it impossible for the LLM to repeat the same incorrect response, as is achieved by purely symbolic tools.

ACKNOWLEDGMENTS

This material is partly supported by the National Science Foundation under Graduate Research Fellowship Grant #1938052. This work has also been partly supported by NSF's FMitF (Formal Methods in the Field) program, under awards 2319500 and 2525087. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Is the SyGuS keyword 'set-feature' supported in CVC4? (issue # 6182). https://github.com/cvc5/cvc5/issues/6182 (2021), gitHub issue, accessed: 2026-01-06

[2] Abdelatty, M., Nouh, M., Rosenstein, J.K., Reda, S.: Pluto: A benchmark for evaluating efficiency of LLM-generated hardware code. arXiv preprint arXiv:2510.14756 (2025)

[3] Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. Formal Methods Syst. Des. **57**(1), 53–86 (2021), https://doi.org/10.1007/s10703-020-00352-2

[4] Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 934–950. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_67

[5] Alur, R., Bodík, R., Juniwal, G., Martin, M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD. pp. 1–17 (2013)

[6] Alur, R., Fisman, D., Padhi, S., Singh, R., Udupa, A.: SyGuS-COMP 2018: Results and analysis. arXiv preprint arXiv:1904.07146 (2019)

[7] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022)

[8] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: cvc4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)

[9] Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)

[10] Biere, A.: The AIGER and-inverter graph (AIG) format version 20071012 (2007)

[11] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners (2020), https://arxiv.org/abs/2005.14165

[12] Bursuc, S., Ehrenborg, T., Lin, S., Astefanoaei, L., Chiosa, I.E., Kukovec, J., Singh, A., Butterley, O., Bizid, A., Dougherty, Q., et al.: A benchmark for vericoding: Formally verified program synthesis. arXiv preprint arXiv:2509.22908 (2025)

[13] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: International Conference on Computer Aided Verification. pp. 334–342. Springer (2014)

[14] Chakraborty, S., Lahiri, S., Fakhoury, S., Lal, A., Musuvathi, M., Rastogi, A., Senthilnathan, A., Sharma, R., Swamy, N.: Ranking LLM-generated loop invariants for program verification. In: Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 9164–9175 (2023)

[15] Chamarthi, H.R., Dillinger, P.C., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving. In: Hardin, D.S., Schmaltz, J. (eds.) Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011. EPTCS, vol. 70, pp. 4–19 (2011), https://doi.org/10.4204/EPTCS.70.1

[16] Chaudhuri, S.: Neurosymbolic program synthesis. Handbook on Neurosymbolic AI and Knowledge Graphs **400**, 532–549 (2025)

[17] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), https://arxiv.org/abs/2107.03374

[18] Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic. vol. 1, pp. 3–50 (1957)

[19] Cosler, M., Hahn, C., Omar, A., Schmitt, F.: Neurosynt: A neuro-symbolic portfolio solver for reactive synthesis. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 45–67. Springer Nature Switzerland, Cham (2024)

[20] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[21] Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: Acl2s: "the ACL2 sedan". In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume. pp. 59–60. IEEE Computer Society (2007), https://doi.org/10.1109/ICSECOMPANION.2007.14

[22] Egolf, D.: PolySemist TACAS 2025 Artifact (2025). https://doi.org/10.5281/zenodo.14618423, https://doi.org/10.5281/zenodo.14618423

[23] Egolf, D., Schultz, W., Tripakis, S.: Efficient Synthesis of Symbolic Distributed Protocols by Sketching . In: FMCAD 2024: Formal Methods in Computer-Aided Design (2024)

[24] Egolf, D., Tripakis, S.: Accelerating Protocol Synthesis and Detecting Unrealizability with Interpretation Reduction. In: 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2025)

[25] Egolf, D., Tripakis, S.: Recursive program synthesis from sketches and mixed-quantifier properties (2026), https://arxiv.org/abs/2601.04045

[26] Egolf, D., Zhou, Y., Tripakis, S.: Can LLMs Perform Synthesis? (2026), https://arxiv.org/abs/2603.20264

[27] Egolf, D., Zhou, Y., Tripakis, S.: Can LLMs Perform Synthesis? (Artifact, FMCAD 2026) (2026). https://doi.org/10.5281/zenodo.21702852

[28] Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 229–239. ACM (2015), https://doi.org/10.1145/2737924.2737977

[29] Finkbeiner, B.: Synthesis of reactive systems. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016)

[30] Golia, P., Roy, S., Meel, K.S.: Program synthesis as dependency quantified formula modulo theory. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI

2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 1894–1900. ijcai.org (2021), https://doi.org/10.24963/ijcai.2021/261

[31] Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Foundations and Trends in Programming Languages **4**(1-2), 1–119 (2017). https://doi.org/10.1561/2500000010

[32] Hong, Q., Aiken, A.: Recursive program synthesis using paramorphisms. Proc. ACM Program. Lang. **8**(PLDI), 102–125 (2024), https://doi.org/10.1145/3656381

[33] Hozzová, P., Amrollahi, D., Hajd ú, M., Kovács, L., Voronkov, A., Wagner, E.M.: Synthesis of recursive programs in saturation. In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I . pp. 154–171. Lecture Notes in Computer Science, Springer (2024), https://doi.org/10.1007/978-3-031-63498-7_10

[34] Huang, D., Qing, Y., Shang, W., Cui, H., Zhang, J.M.: EffiBench: Benchmarking the efficiency of automatically generated code. Advances in Neural Information Processing Systems **37**, 11506–11544 (2024)

[35] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al.: Qwen2.5-Coder technical report. arXiv preprint arXiv:2409.12186 (2024)

[36] Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1. 1. arXiv preprint arXiv:1604.02284 (2016)

[37] Jacobs, S., Pérez, G.A., Abraham, R., Bruyere, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., et al.: The reactive synthesis competition (SYNTCOMP): 2018–2021. International journal on software tools for technology transfer **26**(5), 551–567 (2024)

[38] Jin, P., Huang, D., Li, C., Cheng, S., Zhao, Y., Zheng, X., Zhu, J., Xing, S., Dou, B., Zhang, R., et al.: RealBench: Benchmarking Verilog generation models with real-world IP designs. arXiv preprint arXiv:2507.16200 (2025)

[39] Joshi, H., Ebenezer, A., Sanchez, J.C., Gulwani, S., Kanade, A., Le, V., Radiček, I., Verbruggen, G.: Flame: A small language model for spreadsheet formulas. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 38, pp. 12995–13003 (2024)

[40] Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Finding inductive loop invariants using large language models. arXiv preprint arXiv:2311.07948 (2023)

[41] Kang, M., Liu, M., Hamad, G.B., Suhaib, S.M., Ren, H.: FVEval: Understanding language model capabilities in formal verification of digital hardware. In: 2025 Design, Automation & Test in Europe Conference (DATE). pp. 1–6. IEEE (2025)

[42] Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis , IN, USA, October 26-31, 2013. pp. 407–426. ACM (2013), https://doi.org/10.1145/2509136.2509555

[43] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 316–329. ACM (2010), https://doi.org/10.1145/1806596.1806632

[44] Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (Jun 2002)

[45] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying Rust programs using linear ghost types. Proceedings of the ACM on Programming Languages **7**(OOPSLA1), 286–315 (2023)

[46] Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010), https://doi.org/10.1007/978-3-642-17511-4_20

[47] Li, Y., Parsert, J., Polgreen, E.: Guiding enumerative program synthesis with large language models. In: International Conference on Computer Aided Verification. pp. 280–301. Springer (2024)

[48] Liu, J., Xie, S., Wang, J., Wei, Y., Ding, Y., Zhang, L.: Evaluating Language Models for Efficient Code Generation. arXiv preprint arXiv:2408.06450 (2024)

[49] Liu, M., Pinckney, N., Khailany, B., Ren, H.: VerilogEval: Evaluating large language models for Verilog code generation. In: 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). pp. 1–8. IEEE (2023)

[50] Loughridge, C.R., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M.R.H., Amin, N., Tegmark, M.: Dafny-Bench: A Benchmark for Formal Software Verification. Transactions on Machine Learning Research (2025), https://openreview.net/forum?id=yBgTVWccIx

[51] Lu, Y., Liu, S., Zhang, Q., Xie, Z.: RTLLM: An open-source benchmark for design RTL generation with large language model. In: 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 722–727. IEEE (2024)

[52] Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proc. ACM Program. Lang. **4**(ICFP), 109:1–109:29 (2020), https://doi.org/10.1145/3408991

[53] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (Jan 1980). https://doi.org/10.1145/357084.357090

[54] Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022), https://doi.org/10.1145/3498682

[55] Miltner, A., Wang, Z., Chaudhuri, S., Dillig, I.: Relational synthesis of recursive programs via constraint annotated tree automata. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14683, pp. 41–63. Springer (2024), https://doi.org/10.1007/978-3-031-65633-0_3

[56] Moura, L.d., Ullrich, S.: The Lean 4 theorem prover and programming language. In: International Conference on Automated Deduction. pp. 625–635. Springer (2021)

[57] OpenAI, :, El-Kishky, A., Wei, A., Saraiva, A., Minaiev, B., Selsam, D., Dohan, D., Song, F., Lightman, H., Clavera, I., Pachocki, J., Tworek, J., Kuhn, L., Kaiser, L., Chen, M., Schwarzer, M., Rohaninejad, M., McAleese, N., o3 contributors, Mürk, O., Garg, R., Shu, R., Sidor, S., Kosaraju, V., Zhou, W.: Competitive programming with large reasoning models (2025), https://arxiv.org/abs/2502.06807

[58] Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 619–630. ACM (2015), https://doi.org/10.1145/2737924.2738007

[59] Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS language standard version 2.1. arXiv preprint arXiv:2312.06001 (2023)

[60] Peng, Z., Yin, X., Qian, R., Lin, P., Liu, Y., Zhang, H., Ying, C., Luo, Y.: SolEval: Benchmarking Large Language Models for Repository-level Solidity Smart Contract Generation. In: Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing. pp. 4388–4411 (2025)

[61] Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. SFCS '77 (1977)

[62] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: ACM Symp. POPL (1989)

[63] Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 522–538. ACM (2016), https://doi.org/10.1145/2908080.2908093

[64] Qiu, R., Zeng, W.W., Ezick, J., Lott, C., Tong, H.: How efficient is LLM-generated code? A rigorous & high-standard benchmark. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=suz4utPr9Y

[65] Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting ltlsynt. Formal Methods in System Design **61**(2), 248–289 (2022)

[66] Schmitt, F., Cosler, M., Finkbeiner, B.: Neural circuit synthesis with pre-trained language models. In: First International Workshop on Deep Learning-aided Verification (2023)

[67] Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. In: Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems. vol. 34, pp. 15408–15420. Curran

Associates, Inc. (2021), https://proceedings.neurips.cc/paper_files/paper/2021/file/8230bea7d54bcdf99cdfe85cb07313d5-Paper.pdf
[68] Singh, A., Fry, A., Perelman, A., Tart, A., Ganesh, A., El-Kishky, A., McLaughlin, A., Low, A., Ostrow, A., Ananthram, A., et al.: OpenAI GPT-5 System Card. arXiv preprint arXiv:2601.03267 (2025)
[69] Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 475–495 (oct 2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
[70] Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 281–294. PLDI '05, ACM (2005), http://doi.acm.org/10.1145/1065010.1065045
[71] Sun, C., Agashe, V., Chakraborty, S., Taneja, J., Barrett, C.W., Dill, D.L., Qiu, X., Lahiri, S.K.: ClassInvGen: Class Invariant Synthesis Using Large Language Models . In: Giacobbe, M., Lukina, A. (eds.) AI Verification - Second International Symposium, SAIV 2025 , Zagreb, Croatia, July 21-22, 2025, Proceedings. Lecture Notes in Computer Science, vol. 15947, pp. 64–96. Springer (2025), https://doi.org/10.1007/978-3-031-99991-8_4
[72] cvc5 Team: cvc5 code repository, https://github.com/cvc5/cvc5
[73] Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., Chaudhuri, S.: CLEVER: A curated benchmark for formally verified code generation. arXiv preprint arXiv:2505.13938 (2025)
[74] Tihanyi, N., Jain, R., Charalambous, Y., Ferrag, M.A., Sun, Y., Cordeiro, L.C.: A new era in software security: Towards self-healing software via large language models and formal verification (2024), https://arxiv.org/abs/2305.14752
[75] TLA+ Foundation: GenAI-accelerated TLA+ Challenge (2025), https://foundation.tlapl.us/challenge/index.html, challenge description and results page, accessed 23 January 2026
[76] Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., Song, D.: Verina: Benchmarking Verifiable Code Generation. arXiv preprint arXiv:2505.23135 (2025)
[77] Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods. pp. 54–66. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
[78] Zhuo, T.Y., Vu, M.C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I.N.B., Zhan, H., He, J., Paul, I., et al.: BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. arXiv preprint arXiv:2406.15877 (2024)