Combining LLM Code Generation with Formal Specifications and Reactive Program Synthesis

Anonymous submission

Abstract

In the past few years, Large Language Models (LLMs) have exploded in usefulness and popularity for code generation tasks. However, LLMs still struggle with accuracy and are unsuitable for high-risk applications without additional oversight and verification. In particular, they perform poorly at generating code for highly complex systems, especially with unusual or out-of-sample logic. For such systems, verifying the code generated by the LLM may take longer than writing it by hand. We introduce a solution to reduce the amount of code that needs to be reasoned about by separating the code generation into two parts; one to be handled by an LLM and one to be handled by formal methods-based program synthesis. We develop a benchmark to test our solution and show that it improves the accuracy of LLM-generated code.

Introduction

The potential for LLMs to increase the productivity of software engineers through code-generation has been well-demonstrated through popular benchmarks such as HumanEval (Chen et al. 2021) or SWE-Bench (Jimenez et al. 2023) However, despite their strong code-writing capabilities, the adoption of LLMs in writing code for mission-critical systems hinges on the inability of LLMs to write code with formal correctness guarantees. In particular, using LLMs for the generation of code bases containing potentially hundreds of thousands of lines requires that the generated code then be manually verified, a time-consuming task that negates many of the advantages of LLM code generation.

In this work, we propose that LLM code generation can be combined with the large body of existing work in program synthesis in formal methods (Alur et al. 2013; Gulwani 2011; Jacobs and Bloem 2018), both to increase correctness of generations as well as to decrease the lines of code that must be manually verified. At a high level, we use LLMs to generate formal specifications from natural language, we then use program synthesis to generate code that is "correct-by-construction". A major challenge to this approach is that program synthesis from formal specifications generally targets a small, fixed grammar, meaning we are not able to leverage the flexibility of LLM code generation. Our key insight is to use formal specification languages that allow for "holes" in the generated code that can be later filled in with LLM code generation. In this way, we use program synthesis from formal synthesis only for the generation of the structure of the code base, then we can fill in the details with an LLM in such a way that the structural guarantees are still valid regardless of the output of the LLM.

Specifically, we use Temporal Stream Logic, which allows authors to specify brief logical constraints on a system's behavior to generate reactive systems with complexity surpassing what a maintainer could easily implement, and augmenting it with the flexibility and reasoning capabilities of LLMs. In this way, we open the door to the creation of systems that can dynamically generate trustworthy code for high-risk reactive systems, tackling the issue of trustworthiness of LLMs and the drawback of complex specs of TSL.

In summary, we identify the key contributions of this work as follows:

- Propose a framework for combining formal specification-based program synthesis with LLM code generation to reduce the amount of generated code that must be verified.
- An instantiation of this framework into a code generation pipeline using Temporal Stream Logic.
- An evaluation of our system on a two set of reactive program synthesis benchmarks.

Related Work

LLM Code Generation LLM code generation has advanced significantly to where models are able to outperform humans at competitive programming benchmarks such as HumanEval (Chen et al. 2021). Popular models like GPT-4 (OpenAI et al. 2024) and open source models like CodeLlama (Rozière et al. 2024) have made code generation accessible to researchers and industry to tackle problems that were previously out of scope for traditional program synthesis techniques (Jimenez et al. 2023). In Magicoder, (Wei et al. 2024) find that seeding code generation with randomly sourced code snippets improves the quality of generated code significantly, indicating that using synthesized code to seed generation may be a promising strategy. However, most benchmarks used for code evaluation (Chen et al. 2021; Wei et al. 2024; Rozière et al. 2024) focus on short competitive coding problems, which require relatively few lines of code and are relatively easily verifiable. Benchmarks such

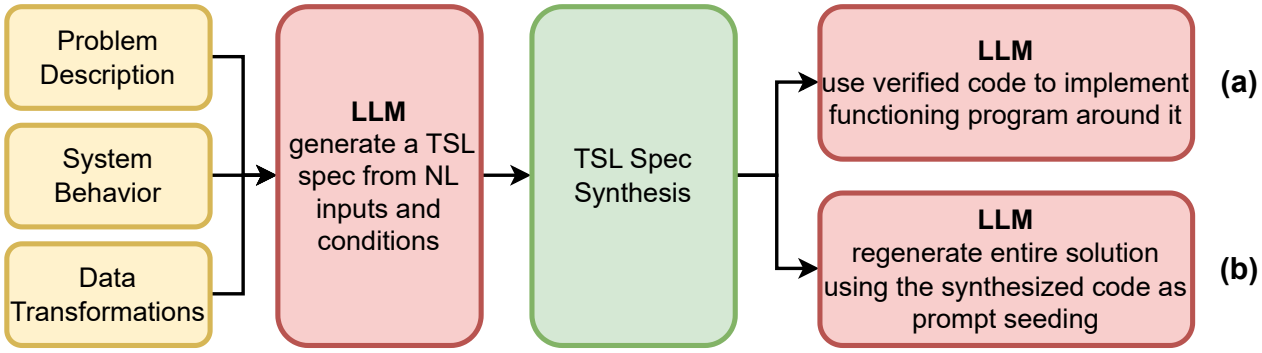


Figure 1: Overview of the full TSL + LLM code generation pipeline. The pipeline turns natural language into executable code. Pipeline **(a)** implements wrapper code to interact with the synthesized code similar to an API. Pipline **(b)** uses the synthesized code to seed an LLM prompt to generate the program described by the NL descriptions.

as SWE-Bench (Jimenez et al. 2023) are larger in scope and are markedly harder for LLMs to solve correctly. In the case that the generated code requires manual verification beyond the provided tests, the use of code generation becomes an exhaustively time-consuming process.

Formal Synthesis The field of Reactive Synthesis has seen tremendous progress since its first formalization as the Church synthesis problem (Church 1962). Most commonly specifications are provided through Linear Temporal Logic (LTL), and the goal is to synthesize a system that reacts to an infinite stream of inputs. LTL aims to place logical, mathematically rigorous guarantees on a system's behavior, to verify an existing system or generate a new one. A primary example of the successful use of LTL for reactive synthesis is the AMBA bus protocol (Bloem, Jacobs, and Khalimov 2014). Since then, the field has seen significant milestones in education (Ma'ayan and Maoz 2023), FPGA game development (Geier et al. 2019), music (Choi, Vazirani, and Santolucito 2021), and interactive animations (Rothkopf et al. 2023). Other formal synthesis systems like Temporal Stream Logic (TSL) (Finkbeiner et al. 2019) provide important advancements, separating data and control, and utilizing function predicate terms which simplify code synthesis but make the final implementation harder for users, who are required to implement the function and predicate terms as well as handle integration of the synthesized code into larger projects (Finkbeiner et al. 2019).

TSL introduces predicate terms, $\tau_P \in \mathcal{T}_P$, which are used to make observations on the environment, and function terms $\tau_F \in \mathcal{T}_F$, which are used to construct output values. These predicate terms enable users to decouple the data and control aspects of a system, encapsulating functionality within functions and predicates that are not pertinent to the specification. As a result, the specification only needs to address the essential parts of the system required to ensure the desired guarantees. In this work, the separation of data and control allows us to leverage the LLM's flexibility for code generation of function and predicate terms, while also allowing Reactive Synthesis with TSL to handle the logical reasoning task of temporal structure that is less well-suited to an LLM.

Using LLMs and Formal Synthesis In pursuit of enhancing the capabilities of formal and LLM systems, researchers have explored combining both approaches. The works of nl2spec (Cosler et al. 2023) and Lang2LTL (Liu et al. 2022), have explored the transformation of natural language descriptions into LTL specifications. These approaches aim to bridge the gap between informal user requirements and formal temporal logic specifications, enabling a more accessible and intuitive method for defining system behaviors and properties. (Rothkopf, Zeng, and Santolucito 2024) explored how reactive synthesis can be leveraged to enforce temporal constraints on content generated by LLMs, but do not explore LLM code generation.

Zero, few-shot learning and in context learning (ICL) are exciting new abilities of powerful LLMs, that enable performance similar to fine-tuned models without costly dataset creation and fine-tuning processes (Brown et al. 2020). Zero and few-shot learning can boost model performance significantly; (Brown et al. 2020) find that models demonstrate an ability to learn about a problem from structured prompts, we believe that the highly structured nature of TSL (Finkbeiner et al. 2019) will help LLMs correctly convert NL to TSL. Additionally, we leverage the ability of models to do ICL and learn from the structure of inputs to boost performance (Min et al. 2022).

System Overview

Our system is a code generation pipeline as shown in Fig. 1, which leverages LLMs to generate TSL specs, and then utilizes correct-by-construction synthesized code to complete critical aspects of data flow control problems such as arbiters or other reactive systems. Our system uses LLMs to overcome the burden of writing TSL specifications and leverages the correct-by-construction nature of synthesized code to reduce the amount of unverified code in mission-critical applications. Moreover, by abstracting complex state behaviors away from the LLM, we can use our pipeline to generate controllers for highly complex multi-agent environments.

Structured NL Prompt Sample

```
You can assume that eventually every
↪  truck will make a request. Further,
↪  guarantee that:

1. for each truck 1..3, if the truck
↪  makes a request, then eventually it
↪  will be given a grant.

2. If the coinflip between truck 2 and
↪  3 resolves to true and truck 1 is
↪  granted the road, then truck 2 will
↪  not be granted the road until truck
↪  3 is granted.

3. If truck 1 is given a grant, then
↪  truck 2 won't be until truck 3 is.
```

(a) Example of a structured NL prompt that is used to prompt the LLM to generate a TSL specification. This example does not include the few-shot preamble used in prompting the LLM to generate (b).

TSL specification Sample

```
# Assumptions: describes inputs from
↪  the environment
always assume {
  F (r t);
}

# Guarantees: describes how the agent
↪  react to those inputs
always guarantee {
    (r 1) -> F ([ g <- 1 ]);
    (r 2) -> F ([ g <- 2 ]);
    (r 3) -> F ([ g <- 3 ]);
    ((p 2 3) && [ g <- 1 ] ->  ! ([ g
    ↪  <- 2 ]) W [ g <- 3 ]);
    ([ g <- 1 ] -> ((! ([ g <- 2 ]) W [
    ↪  g <- 3 ])));
}
```

(b) Example of a TSL specification that describes an arbiter that can handle three requesters. This specification is semantically identical to the NL description in (a).

Figure 2: Examples of the TSL specification interface in both natural language and formal logic.

Our pipeline is composed of the following steps:

1. Inputs:
   (a) A high-level, natural language summary of the problem (Problem Description in Fig. 1).
   (b) A more detailed, natural language description of the most important assumptions and guarantees needed in the spec (System Behavior in Fig. 1).
   (c) A separation of data and control in the form of function and predicate terms. The specification should use these to encapsulate logic not relevant to the assumptions and guarantees (Data Transformations in Fig. 1).
2. Using prompt engineering, we utilize an LLM to convert the NL inputs into a TSL specification
3. We use formal methods-based program synthesis to solve the TSL specification and synthesize correct-by-construction code.
4. Using an LLM, we integrate the formally verified code into the larger codebase or project. We capitalize on the unique nature of TSL that separates data and control to generate lower-risk encapsulating logic using an LLM to integrate the verified code, thereby reducing the burden of verification on human developers.
5. Alternatively, we adopt the prompt seeding technique of Magicoder (Wei et al. 2024) and use the synthesized code to request the LLM to generate from the ground up a working solution. When implementing real world complex decision systems, this approach will seed the LLM prompt with relevant code for the control logic that is guaranteed to be correct, potentially boosting the rate of generating correct code for the whole system.

Natural Language (NL) to TSL

An important part of our pipeline revolves around an NL to TSL conversion handled by the LLM. We leverage ICL and few-shot prompting to create TSL prompts from detailed NL inputs. We use a prompting template that contains several NL to TSL examples, as well as explicit explanations about TSL terms and statements. We find that GPT-4 (OpenAI et al. 2024) can perform very well using few shot prompting. This is likely due to the highly structured and logical nature of TSL, with which models can perform ICL exceptionally well (Min et al. 2022; Brown et al. 2020).

TSL's structure allows for a exceptionally clean logical separation of data and control in a program. As shown in Fig. 3, IO streams are separate from the reactive system, as seen in Fig. 2a and Fig. 2b, we define function and predicate terms, r, g to handle state changes and data flow in the system, giving greater flexibility to the LLM in creating TSL specifications as well as facilitating integration into full programs. The flexibility that function and predicate terms give TSL specifications (Finkbeiner et al. 2019), is likely a factor that facilitates ICL in our tasks, since the LLM can leverage its coding and reasoning strengths to a greater extent in the NL to TSL step (Min et al. 2022; Brown et al. 2020). The utility of the function and predicate terms then carries over to the final code implementation step shown in Fig. 1, where the LLM can leverage the flexibility of the data control separation in the synthesized code to easily generate wrapper code for the synthesized reactive system. Due to the pure functional (side-effect free) nature of the wrapper code, function and predicate terms resolve to functions with clearly defined functionality; this structure can help to make verification a more tractable task.

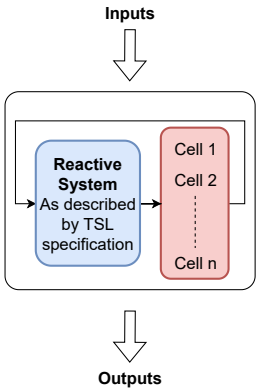


Figure 3: Overview of a reactive system as implemented in a formal TSL specification.

Formal Verification Assisted Code Generation

Our pipeline aims to reduces the number of lines of unverified code in projects by removing the burden of generation from the LLM and moving it to the formal synthesis procedure. We do this through the introduction of function and predicate terms that manipulate data at a single point in time. These predicate terms enable users to decouple the data and control (temporal) aspects of a system, encapsulating data transformations within functions and predicates that are not pertinent to the temporal specification (Finkbeiner et al. 2019). As a result, the scope of the specification is reduced to only address the temporal aspect of the system, as opposed to, for example LTL, where both data and control are the captured by the specification and are the responsibility of the synthesis engine.

This is achievable due to TSL's introduction of specification-level predicate and function terms. Predicate terms, $\tau_P \in \mathcal{T}_P$, are used to make observations on the environment, and function terms, $\tau_F \in \mathcal{T}_F$, are used to construct output values, where $\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$ is an input stream or cell value, and $\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$ is an output stream or cell value. Together, all the available predicate names $\mathtt{p}$ and all the available function names $\mathtt{f}$ form the set of function symbols $\mathbb{F}$. A TSL formula describes a system that consumes input $\mathcal{I} = \mathbb{I} \cup \mathbb{C}$ and produces output $\mathcal{O} = \mathbb{O} \cup \mathbb{C}$ as shown in Eq. (1). As shown in Fig. 3 the reactive system converts an input stream into an output stream, using cells to track states.

$$\begin{aligned} \tau_P &:= \mathtt{p}\ \tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1} \\ \tau_F &:= \mathtt{s_i} \mid \mathtt{f}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1} \\ \varphi &:= \tau_P \mid [\mathtt{s_o} \leftarrowtail \tau_F] \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \end{aligned} \tag{1}$$

The realizability problem of TSL is stated as follows: given a TSL formula φ, is there a strategy $\sigma \in \mathcal{I}^+ \to \mathcal{O}$ mapping a finite input stream (since the beginning of time) to an output (at each particular timestep), such that for any infinite input stream $\iota \in \mathcal{I}^\omega$, and every possible interpretation of the function symbols (where an interpretation is some concrete implementation in code) $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$, the execution of that strategy over the input $\sigma \wr \iota$ satisfies φ, i.e.,

$$\exists \sigma \in \mathcal{I}^+ \to \mathcal{O}.\ \forall \iota \in \mathcal{I}^\omega.\ \forall \langle\cdot\rangle : \mathbb{F} \to \mathcal{F}.\ \sigma \wr \iota, \iota \models_{\langle\rangle} \varphi \tag{2}$$

If such a strategy σ exists, we say that σ realizes φ. The key insight here is that in TSL we are universally quantifiying over implementations of predicate and function terms. The specification φ only describes a temporal relation of predicate evaluations to function applications–abstracting away from what these predicates and functions do to any underlying data. In TSL synthesis, this model σ can be turned into a block of program code that describes a Mealy machine (Mealy 1955), where the transitions represent function and predicate terms. These assurances allow us to treat synthesized code as verified in large codebases. The user only needs to manually check the correctness of the specification is correct.

Simplifying Long Context Problems

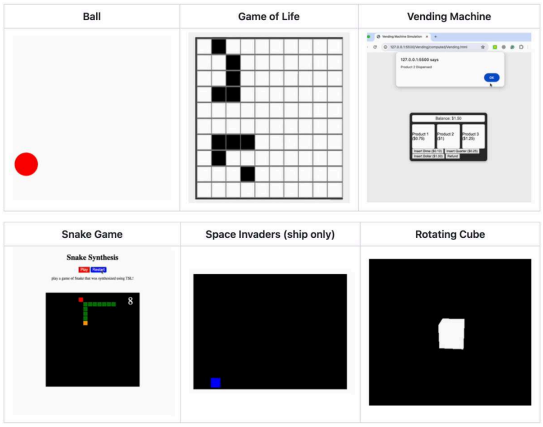
Using TSL specifications allows us to shorten the context of NL instructions describing reactive systems.

TSL's data and control separation through its universal quantification of function and predicate terms is particularly well-suited for long-context LLM generation. TSL lets us encode complex conditional statements in a compact syntactical representation. In this way, the LLM generated TSL specifications are significantly easier to verify than long LLM-generated code bases, increasing efficiency and trust in the code.

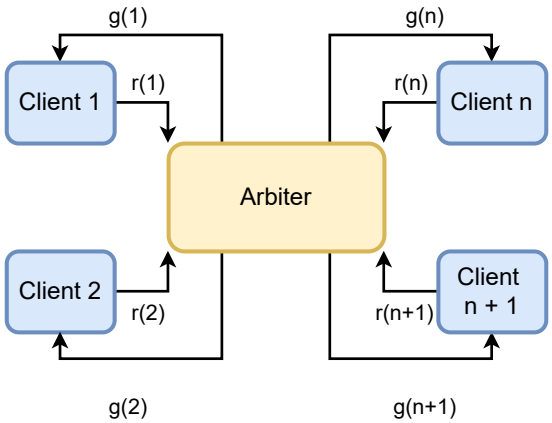
This separation enables developers to first focus on when their system should execute certain behaviors, and leave the question of how those behaviors should be implemented for a later step in the development process. Traditionally, the control is synthesized from TSL and the end-user implements the data transformations manually. We propose automating both processes using LLMs.

Evaluation

We evaluate our approach in two different contexts. First, we seek to quantify the improvements of our TSL-enhanced generation over traditional LLM-only code generation. We measure and compare the number of unverified lines of code written by either method over a series of project-driven benchmarks whose goal is to implement simple reactive programs as demonstrated in 4(a). Next, we compare the benefits of using our TSL pipeline in generating solutions to complex arbiter problems as shown in 4(b); we seek to progressively explore the limits of LLM-only generated arbiters as opposed to TSL pipelines. We depart from traditional code generation testing frameworks (Chen et al. 2021) and design our tasks to be very hard for LLMs to solve, following (Jimenez et al. 2023) in SWE-Bench.



(a) Project Oriented Benchmark



(b) Arbiter Benchmark

Figure 4: Six different project-oriented coding benchmarks that require working implementation of reactive applications and games.

Testing Methodology

Task Oriented Code Generation

The task-oriented code generation metric seeks to quantify the reduction in lines of unverified code present in a codebase when using TSL pipelines. We count the lines of code written for a pass at k condition to create a working implementation of the program for both the TSL pipeline and the LLM-only approach. To ensure consistency across both approaches, we use the same structured NL prompt shown in Fig. 1 in both runs, prompting the LLM to generate a working implementation from scratch in one case, and a TSL spec in the other. In our experiments, we include runs where the TSL specification fails to compile. We do this to explore the effects of changing the LLM architecture on few-shot prompting with examples. In real-world applications, rejecting unsynthesizable TSL specifications and using techniques like reprompting can lead to much better success rates. As shown in Fig 1(b), we take inspiration from (Wei et al. 2024) who show that seeding LLM code generation with random code significantly improves output. We expect that using verified and task-specific code to further improve the outputs of the pipeline over LLM-only generation.

We implement this task with a full generation objective, that tracks the total lines of code needed to generate a working implementation of the task. As such we are able to demonstrate differences in the lines of unverified code introduced into codebases as well as examine the success rate of various methods when generating complex programs with many lines of code.

Complex Arbiter Problem Solution Generation

We evaluate the improvements of our approach on a scaling arbiter benchmark as shown in 4(b). This task separates data and control in a resource management game, in which the generated code must track resource requests and requesters and follow these rules:

- Every request of a client is eventually granted by the arbiter.
- The arbiter never two grants at the same time.
- The arbiter only grants a client i if it has an open request.
- A client may only pose a request if it has no open request.

This task is designed to leverage the verified nature of the synthesized lines. Since the number of lines required to design an arbiter scales with the number of conditions in the arbiter, we seek to examine the effectiveness of TSL-enhanced pipelines in generating working arbiters, as well as the ability of TSL to cut down on the number of unverified lines introduced into the code base.

Results

Trusted Code and Prompt Seeding

The first task measures the reduction in lines of unverified code as well as the success rate of full generation both via pipeline and LLM-only methods. Our results indicate that the two ways of using the TSL pipeline both have advantages and disadvantages. On the one hand, using the TSL pipeline without prompt seeding as shown in Fig. 1(a) results in a large portion of verified lines implementing the system which do not need to be verified. As seen in Table 1 the given benchmark tasks are too simple to leverage this key benefit of the pipeline. We therefore demonstrate that the pipeline shown in 1(b) can beat a GPT-4 (OpenAI et al. 2024) baseline at generating complex programs. On challenging tasks like implementing a working implementation of Conway's game of life, the prompt seeding approach achieves a 0.80 success rate as opposed to GPT-4's 0.40 success rate at 15 tries.

The rather ambiguous results of the pipeline shown in Fig 1(a) suggests that even longer and more complex problems are needed to leverage the full potential of formally verified

Table 1: Task Oriented Code Generation Performance for TSL pipeline, LLM only code generation and prompt seeding with synthesized code. We compare success rate (sr), average lines of code, and the percentage of unverified lines of code across these tasks.

Tasks	sr 5	sr 10	sr 15	Avg. Total Lines	Avg. # Unverified Lines
Ball_TSL	0.20	0.10	0.13	249	78
Game of Life_TSL	0.60	0.50	0.73	124	98
Vending Machine_TSL	1.00	1.00	**1.00**	192	71
Space Invaders_TSL	0.00	0.00	0.06	88	68
Rotating Cube_TSL	0.40	0.40	0.33	108	75
Ball_LLM	0.20	0.10	0.13	55	-
Game of Life_LLM	0.40	0.50	0.40	87	-
Vending Machine_Regen	1.00	1.00	**1.00**	70	-
Space Invaders_LLM	0.40	0.30	**0.27**	68	-
Rotating Cube_Regen	0.80	0.60	**0.80**	92	-
Ball_Regen	1.00	1.00	**1.00**	49	-
Game of Life_Regen	0.60	0.70	**0.80**	98	-
Vending Machine_Regen	1.00	1.00	**1.00**	63	-
Space Invaders_Regen	0.00	0.00	0.06	68	-
Rotating Cube_Regen	0.80	0.60	**0.80**	45	-

code in codebases, which we explore further in our other experiments. However, we are able to demonstrate strong improvements at complex tasks when using the prompt seeding approach of the pipeline shown in Fig 1(b). Nonetheless, Table 1 shows that pipeline (a) can match GPT-4 only generation in some tasks, even beat it by 0.33 at Conway's game of life. Our results therefore indicate that using LLMs to generate code deployed in high-risk decision making environments, requiring hundreds or possibly thousands of guarantees and logical conditions, may very well stand to benefit from pipelines that incorporate TSL.

Our method does introduce certain drawbacks, especially through the TSL specification. In certain cases, the LLM was unable to produce a synthesizable TSL specification which led to poor performance compared to LLM only code generation. This weak performance demonstrates the limits of few shot prompting when requiring the LLM to generate a syntactically complicated and unknown language.

Table 2: Arbiter Problem Model Performance Comparison on Success Rate (sr). We run each trial 25 times and show the number of verified synthesized lines as well as the number of generated lines to use the synthesized system.

Metric	**Condition Count**		
	10	**20**	**30**
TSL pipeline (sr)	61.5%	45.4%	16.7%
LLM pipeline (sr)	0%	0%	0%
Average line count (TSL)	22	31	40
Average line count (Synth)	$10,130$	$10,130$	$10,310$

Solving Long Context Problems

Our results for this task indicate that the TSL enhanced pipeline shown in Fig 1(a) is able to enhance the performance of the LLM. Crucially, the success rate of GPT-4 at these arbiter problems is 0, whilst the TSL pipeline is able to solve between 62% and 17% of arbiter problems with $10,\ 20$ or 30 conditions. TSL allows the LLM to solve the problem, and this solution is practical, because the number of generated lines does not grow to the point that each cannot be verified by hand. As shown in Table 2, the number of generated lines grows linearly with the number of conditions; the interacting requirements do not create exponential complexity that would cause it to be impractical to review.

These results demonstrate how using TSL-enhanced generation can introduce the advantages of LLM code generation to a whole new field of complex decision systems.

Conclusion

In this paper, we demonstrates the possibility of building a code generation pipeline by moving key system logic to formal methods-based program synthesis without sacrificing the strong code-generating capabilities of state-of-the-art LLMs. Additionally, we show that, when not concerned with verification, LLM code generation can be significantly improved when using TSL-based prompt seeding.

In conclusion, our work has demonstrated that adding formal verification methods can significantly enhance and improve traditional code generation pipelines. Our results show that across various tasks, we can reduce the lines of unverified code in applications, making this pipeline more desirable for high-risk applications. Moreover, temporal logic allows this pipeline to generate controllers for highly complex arbitration problems. Our pipeline relies on ICL and few-shot prompting to convert NL to TSL, opening interesting future work into fully integrated and fine-tuned pipelines.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. *Syntax-guided synthesis*. IEEE.

Bloem, R.; Jacobs, S.; and Khalimov, A. 2014. Parameterized synthesis case study: AMBA AHB (extended version). *arXiv preprint arXiv:1406.7608*.

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.; and Amodei, D. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165.

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.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Choi, W.; Vazirani, M.; and Santolucito, M. 2021. Program synthesis for musicians: A usability testbed for temporal logic specifications. In *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings 19*, 47–61. Springer.

Church, A. 1962. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, volume 1962, 23–35.

Cosler, M.; Hahn, C.; Mendoza, D.; Schmitt, F.; and Trippel, C. 2023. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, 383–396. Springer.

Finkbeiner, B.; Klein, F.; Piskac, R.; and Santolucito, M. 2019. Temporal stream logic: Synthesis beyond the bools. In *International Conference on Computer Aided Verification*. Springer.

Geier, G.; Heim, P.; Klein, F.; and Finkbeiner, B. 2019. Syntroids: Synthesizing a game for fpgas using temporal logic specifications. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, 138–146. IEEE.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1): 317–330.

Jacobs, S.; and Bloem, R. 2018. The 5th reactive synthesis competition—SYNTCOMP 2018. In *SYNT workshop at FLoC*.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Liu, J. X.; Yang, Z.; Schornstein, B.; Liang, S.; Idrees, I.; Tellex, S.; and Shah, A. 2022. Lang2ltl: translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*.

Ma'ayan, D.; and Maoz, S. 2023. Using Reactive Synthesis: An End-to-End Exploratory Case Study. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 742–754. IEEE.

Mealy, G. H. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5): 1045–1079.

Min, S.; Lyu, X.; Holtzman, A.; Artetxe, M.; Lewis, M.; Hajishirzi, H.; and Zettlemoyer, L. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? arXiv:2202.12837.

OpenAI; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; Avila, R.; Babuschkin, I.; Balaji, S.; Balcom, V.; Baltescu, P.; Bao, H.; Bavarian, M.; Belgum, J.; Bello, I.; Berdine, J.; Bernadett-Shapiro, G.; Berner, C.; Bogdonoff, L.; Boiko, O.; Boyd, M.; Brakman, A.-L.; Brockman, G.; Brooks, T.; Brundage, M.; Button, K.; Cai, T.; Campbell, R.; Cann, A.; Carey, B.; Carlson, C.; Carmichael, R.; Chan, B.; Chang, C.; Chantzis, F.; Chen, D.; Chen, S.; Chen, R.; Chen, J.; Chen, M.; Chess, B.; Cho, C.; Chu, C.; Chung, H. W.; Cummings, D.; Currier, J.; Dai, Y.; Decareaux, C.; Degry, T.; Deutsch, N.; Deville, D.; Dhar, A.; Dohan, D.; Dowling, S.; Dunning, S.; Ecoffet, A.; Eleti, A.; Eloundou, T.; Farhi, D.; Fedus, L.; Felix, N.; Fishman, S. P.; Forte, J.; Fulford, I.; Gao, L.; Georges, E.; Gibson, C.; Goel, V.; Gogineni, T.; Goh, G.; Gontijo-Lopes, R.; Gordon, J.; Grafstein, M.; Gray, S.; Greene, R.; Gross, J.; Gu, S. S.; Guo, Y.; Hallacy, C.; Han, J.; Harris, J.; He, Y.; Heaton, M.; Heidecke, J.; Hesse, C.; Hickey, A.; Hickey, W.; Hoeschele, P.; Houghton, B.; Hsu, K.; Hu, S.; Hu, X.; Huizinga, J.; Jain, S.; Jain, S.; Jang, J.; Jiang, A.; Jiang, R.; Jin, H.; Jin, D.; Jomoto, S.; Jonn, B.; Jun, H.; Kaftan, T.; Łukasz Kaiser; Kamali, A.; Kanitscheider, I.; Keskar, N. S.; Khan, T.; Kilpatrick, L.; Kim, J. W.; Kim, C.; Kim, Y.; Kirchner, J. H.; Kiros, J.; Knight, M.; Kokotajlo, D.; Łukasz Kondraciuk; Kondrich, A.; Konstantinidis, A.; Kosic, K.; Krueger, G.; Kuo, V.; Lampe, M.; Lan, I.; Lee, T.; Leike, J.; Leung, J.; Levy, D.; Li, C. M.; Lim, R.; Lin, M.; Lin, S.; Litwin, M.; Lopez, T.; Lowe, R.; Lue, P.; Makanju, A.; Malfacini, K.; Manning, S.; Markov, T.; Markovski, Y.; Martin, B.; Mayer, K.; Mayne, A.; McGrew, B.; McKinney, S. M.; McLeavey, C.; McMillan, P.; McNeil, J.; Medina, D.; Mehta, A.; Menick, J.; Metz, L.; Mishchenko, A.; Mishkin, P.; Monaco, V.; Morikawa, E.; Mossing, D.; Mu, T.; Murati, M.;

Murk, O.; Mély, D.; Nair, A.; Nakano, R.; Nayak, R.; Neelakantan, A.; Ngo, R.; Noh, H.; Ouyang, L.; O'Keefe, C.; Pachocki, J.; Paino, A.; Palermo, J.; Pantuliano, A.; Parascandolo, G.; Parish, J.; Parparita, E.; Passos, A.; Pavlov, M.; Peng, A.; Perelman, A.; de Avila Belbute Peres, F.; Petrov, M.; de Oliveira Pinto, H. P.; Michael; Pokorny; Pokrass, M.; Pong, V. H.; Powell, T.; Power, A.; Power, B.; Proehl, E.; Puri, R.; Radford, A.; Rae, J.; Ramesh, A.; Raymond, C.; Real, F.; Rimbach, K.; Ross, C.; Rotsted, B.; Roussez, H.; Ryder, N.; Saltarelli, M.; Sanders, T.; Santurkar, S.; Sastry, G.; Schmidt, H.; Schnurr, D.; Schulman, J.; Selsam, D.; Sheppard, K.; Sherbakov, T.; Shieh, J.; Shoker, S.; Shyam, P.; Sidor, S.; Sigler, E.; Simens, M.; Sitkin, J.; Slama, K.; Sohl, I.; Sokolowsky, B.; Song, Y.; Staudacher, N.; Such, F. P.; Summers, N.; Sutskever, I.; Tang, J.; Tezak, N.; Thompson, M. B.; Tillet, P.; Tootoonchian, A.; Tseng, E.; Tuggle, P.; Turley, N.; Tworek, J.; Uribe, J. F. C.; Vallone, A.; Vijayvergiya, A.; Voss, C.; Wainwright, C.; Wang, J. J.; Wang, A.; Wang, B.; Ward, J.; Wei, J.; Weinmann, C.; Welihinda, A.; Welinder, P.; Weng, J.; Weng, L.; Wiethoff, M.; Willner, D.; Winter, C.; Wolrich, S.; Wong, H.; Workman, L.; Wu, S.; Wu, J.; Wu, M.; Xiao, K.; Xu, T.; Yoo, S.; Yu, K.; Yuan, Q.; Zaremba, W.; Zellers, R.; Zhang, C.; Zhang, M.; Zhao, S.; Zheng, T.; Zhuang, J.; Zhuk, W.; and Zoph, B. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Rothkopf, R.; Cui, A. L.; Zeng, H. T.; Sinha, A.; and Santolucito, M. 2023. Towards the Usability of Reactive Synthesis: Building Blocks of Temporal Logic. In *Plateau Workshop*.

Rothkopf, R.; Zeng, H. T.; and Santolucito, M. 2024. Enforcing Temporal Constraints on Generative Agent Behavior with Reactive Synthesis. *arXiv preprint arXiv:2402.16905*.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; Rapin, J.; Kozhevnikov, A.; Evtimov, I.; Bitton, J.; Bhatt, M.; Ferrer, C. C.; Grattafiori, A.; Xiong, W.; Défossez, A.; Copet, J.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and ZHANG, L. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *Forty-first International Conference on Machine Learning*.

AAAI Reproducibility Checklist

This paper:

- Includes a conceptual outline and/or pseudocode description of AI methods introduced (**yes)**
- Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (**yes**)
- Provides well marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper (**yes**)

Does this paper make theoretical contributions? (**no**)

If yes, please complete the list below.

- All assumptions and restrictions are stated clearly and formally. (**NA**)
- All novel claims are stated formally (e.g., in theorem statements). (**NA**)
- Proofs of all novel claims are included. (**NA**)
- Proof sketches or intuitions are given for complex and/or novel results. (**NA**)
- Appropriate citations to theoretical tools used are given. (**NA**)
- All theoretical claims are demonstrated empirically to hold. (**NA**)
- All experimental code used to eliminate or disprove claims is included. (**NA**)

Does this paper rely on one or more datasets? (**yes**)

If yes, please complete the list below.

- A motivation is given for why the experiments are conducted on the selected datasets (**yes**)
- All novel datasets introduced in this paper are included in a data appendix. (**yes**)
- All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (**yes**)
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. (**yes**)
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available. (**yes**)
- All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisfactory. (**NA**)

Does this paper include computational experiments? (**yes**)

If yes, please complete the list below.

- Any code required for pre-processing data is included in the appendix. (**yes**).
- All source code required for conducting and analyzing the experiments is included in a code appendix. (**yes**)
- All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. (**yes**)
- All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (**yes**)
- If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results. (**NA**: the experiment relies on OpenAI, which cannot be guaranteed not to change slightly over time)
- This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks. (**yes**)
- This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics. (**yes**)
- This paper states the number of algorithm runs used to compute each reported result. (**yes**)
- Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information. (**yes**)
- The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank). (**yes**)
- This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments. (**NA**)
- This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting. (**NA**)