LLM-based Scheme for Synthesis of Formal Verification Algorithms*

Itay Cohen and Doron Peled

Bar Ilan University, Ramat Gan 52900, Israel

Abstract. The research of Large Language Models (LLMs) has significant ground to cover in the context of formal verification. In this work, we present a methodology that aims to increase the reliability of code synthesized through the use of LLMs. Our approach capitalizes on the intrinsic knowledge embedded within LLMs to achieve a more reliable code synthesis. We specifically illustrate the possibility of teaching model checking and runtime verification (RV) algorithms through our approach. Our experiments demonstrate that LLMs grasp the concept of dynamic programming, allowing them to synthesize code for these verification tasks with minimal guidance.

1 Introduction

The emergence of large language models has raised significant expectations for their ability to automate programming by simply specifying the desired code requirements. Such capabilities have already been demonstrated with tools that are based on powerful LLMs such as Copilot [15]. Performing code generation tasks is also possible using generic instruction-based LLMs such as ChatGPT [13] and Gemini [2]. The ability to instruct such a system to synthesize code is largely influenced by the LLMs being trained on extensive corpora, including code from various repositories that typically come with textual descriptions. One of the main shortcomings of such a use is that it does not guarantee to synthesize a correct program as a classical realization of specification methods [10,11]. In fact, in general, the use of LLMs involves some inherent suspicion, e.g., due to the hallucination phenomenon, where LLMs tend to provide "some" output even when their knowledge is limited. Currently, the ability to perform nontrivial deduction by LLMs is also limited. Despite these concerns, the use of LLMs to produce code is widely prevailing. As a result, the use of LLM-based code generation tools should be done with caution, and possibly hand-in-hand with understanding the limitations and with appropriate testing of the code.

Our proposed methodology to synthesize code using LLMs is to identify a programming pattern, in the case of this paper, that of dynamic programming [4], which is simple enough to describe in simple terms and common enough to be

* This research was supported by the Israel Science Foundation grant "Validating and controlling software and hardware systems assisted by machine learning" No. 2454/23.

used in code that is likely to be present in common repositories. This assumes that the training phase of the LLM likely encompassed numerous instances of dynamic programming-based code. Such identification allows for familiarizing the LLM with the general schemes of programs that interest us. This is done by incorporating the general scheme into a prompt. From there, with simple incremental steps, we are capable of obtaining the synthesis of code for two formal methods algorithms: model checking and runtime verification, both subscribing to the dynamic programming scheme.

Breaking the program synthesis problem in this way allows confirming through the use of the LLM that it has accurately detected the programming pattern by reviewing its response. In fact, already in our preliminary experiments, we observed that the LLM immediately identified the dynamic programming pattern, even though we did not explicitly use this term when describing it as a programming pattern. Such feedback is important before continuing with further steps of the code construction.

More specifically, the dynamic programming pattern involves an algorithm where components are structured within a graph, with each component possessing values that impact and are affected by neighboring components. These values are continuously updated in an iterative process until convergence is achieved. One class of problems that are related to such a programming scheme are algorithms that analyze execution of systems such as model checking, runtime verification and statistical calculations. Reinforcement learning also aligns with this pattern, especially via the *Bellman equation* calculation, though it is not the focus of our discussion in this work.

After prompting the LLM with the general programming scheme, we can use a second prompt that introduces the formal verification task. The expected solution for the task is expected to rely on the dynamic programming pattern mentioned before. A runtime verification monitor observes an execution of a system and provides a verdict on whether it satisfies a given past time linear temporal logic specification [9]. Each time a new event is observed, a summary of the sequence of events seen so far is updated. In a sense, we can think about this as a linear graph, where a new node is generated with each inspected event; the new node is a successor of (i.e., has an incoming edge from) the previously generated node. In fact, previous nodes in the graph are redundant and can be removed as soon as the new node is constructed.

A model checking algorithm for the temporal logic CTL [5] can also be described in terms of updating nodes in a directed graph of the global states constructed for a modeled system. States of the program can be represented using graph nodes, with edges connecting states with their immediate successors. The updates are performed in phases, where nodes are updated based on the values of their neighbors. For example, we can check from which states some erroneous situation, e.g., buffer overflow, is reachable. In an initial phase, the nodes in which buffer overflow occurs are marked. Then, in each phase, we mark additional nodes that are incoming neighbors of nodes that were already marked in previous phases. In this way, we mark increasingly longer paths from which such error is reachable. We terminate when a phase where no update occurs. Similar to model checking, algorithms that analyze probabilistic systems [3] also subscribe to a similar dynamic programming pattern, through Bellman equation [14].

In a preliminary evaluation, we utilized GPT-4 to apply our code synthesis methodology to both runtime verification and model checking tasks, based on the dynamic programming scheme.

2 Preliminaries

2.1 Computation Tree Logic

Computation Tree Logic (CTL) is a temporal logic that allows reasoning about the executions of a system. CTL extends propositional logic with temporal operators and quantifiers that express properties over computation trees, which represent all possible execution paths of a system. It facilitates the specification of complex system behaviors across multiple execution paths.

Syntax The syntax of CTL formulas is given by the following grammar:

$$\phi ::= true \mid q \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid AX\varphi \mid$$

$$EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A[\varphi U \psi] \mid E[\varphi U \psi]$$

where q belongs to a finite set A of Boolean Boolean propositions. The logic CTL has the following informal meaning. The operators $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ have the same meaning as in propositional calculus. The *state* operators A and E denote "for all paths" and "there exists a path" respectively, and the *path* operators include X (next), F (eventually), G (globally), and U (until). The operator X indicates that a property must hold in the very next state. The F operator expresses that a condition will become true at some point in the future, while G ensures that a property remains true in all future states along the path. The U operator is used to denote that a property φ holds continuously until another property ψ holds. The syntax restricts the state and path operators to alternate.

Semantics The semantics of CTL is defined with respect to a Kripke structure $M = (S, R, L)$, where S is a set of states, $R \subseteq S \times S$ is a transition relation, and L is a labeling function mapping each state to a set of atomic propositions that hold in that state. A Kripke structure can be represented as a directed graph. A *path* in this structure is defined as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$, where each s_i is a state in S and each pair (s_i, s_{i+1}) belongs to the transition relation R.

A CTL formula φ is interpreted over a state s in a model M. The satisfaction relation $(M, s) \models \varphi$ is defined inductively as follows:

- $(M, s) \models true$ is always true.
- $(M, s) \models p$ iff $p \in L(s)$.
- $(M, s) \models \neg\phi$ iff it is not the case that $(M, s) \models \phi$.
- $(M, s) \models \phi \; op \; \psi$ iff $(M, s) \models \phi \; op \; (M, s) \models \psi$, when $op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

- $(M,s) \models AX\phi$ iff for all states t such that $(s,t) \in R$, we have $(M,t) \models \phi$.
- $(M,s) \models EX\phi$ iff there exists a state t such that $(s,t) \in R$ and $(M,t) \models \phi$.
- $(M,s) \models AF\phi$ iff for all paths $\pi = s_0 s_1 s_2 \dots$ starting at s, there exists $i \geq 0$ such that $(M,s_i) \models \phi$.
- $(M,s) \models EF\phi$ iff there exists a path $\pi = s_0 s_1 s_2 \dots$ starting at s and an $i \geq 0$ such that $(M,s_i) \models \phi$.
- $(M,s) \models AG\phi$ iff for all paths $\pi = s_0 s_1 s_2 \dots$ starting at s, for all $i \geq 0$, $(M,s_i) \models \phi$.
- $(M,s) \models EG\phi$ iff there exists a path $\pi = s_0 s_1 s_2 \dots$ starting at s, such that for all $i \geq 0$, $(M,s_i) \models \phi$.
- $(M,s) \models A[\phi U \psi]$ iff for all paths $\pi = s_0 s_1 s_2 \dots$ starting at s, there exists $i \geq 0$ such that $(M,s_i) \models \psi$ and for all $j < i$, $(M,s_j) \models \phi$.
- $(M,s) \models E[\phi U \psi]$ iff there exists a path $\pi = s_0 s_1 s_2 \dots$ starting at s, where there exists $i \geq 0$ such that $(M,s_i) \models \psi$ and for all $j < i$, $(M,s_j) \models \phi$.

2.2 Model Checking of Computation Tree Logic Formulas

In the context of formal verification, model checking [6, 12] stands as a central technique for verifying that a system model satisfies a given specification expressed in temporal logic. CTL model checking involves verifying whether a state in a model satisfies a CTL formula. The core idea is to recursively evaluate the formula over the structure of the model, which is typically represented as a Kripke structure.

The algorithm performs in phases. It marks nodes, representing states, with the subformulas that hold in them (if a node is not marked by a subformula, then it satisfies its negation). Initially, it marks the nodes according to the atomic propositions in the CTL formula that are satisfied. Then, the nodes are marked based on more complex subformulas. That is, if φ is a subformula of ψ, then the phases for φ must terminate before the phases for ψ start. Updating nodes with respect to a temporal subformula can take multiple phases, where in each phase, the marking of a node can be affected by the marking of neighboring nodes in previous phases. The switch from a marking node with φ to ψ happens after a phase where no further nodes are marked by ψ occurs. For example, when $\psi = \varphi_1 \wedge \varphi_2$, after marking nodes with φ_1 and φ_2, a new phase marks the nodes already marked by both φ_1 and φ_2 by φ. When $\psi = EF\varphi$, after marking the nodes with φ, a sequence of phases mark nodes with ψ. First, nodes marked by φ are also marked by ψ. Then, in each subsequent phase, each node that has at least one successor marked in a previous phase by ψ is also marked by ψ. Thus, in each such phase, we mark by ψ nodes with a longer path to a node that satisfies φ than in the previous phase. This terminates when no new marking by ψ occurs. The full algorithm can be found in [6].

2.3 Propositional Past Time Linear Temporal Logic

Propositional past time linear temporal logic (PLTL) is a specification formalism that allows expressing safety [1] properties of executions that include atomic propositions. The restriction to past time allows interpreting the formulas on finite traces.

Syntax The formulas of propositional past time linear temporal logic are defined using the following grammar:

$$\varphi ::= \mathit{true} \mid q \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid \ \varphi \, \mathcal{S} \, \psi \mid \ \ominus \varphi \mid \ \diamondminus \varphi \mid \ \boxminus \varphi$$

where the symbol q belongs to a finite set of Boolean propositions. The operators $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ have the same meaning as in propositional calculus.

The temporal operators have the following informal meaning: the formula $(\varphi \ \mathcal{S} \ \psi)$, which reads as φ *since* ψ, means that ψ holds in some prefix of the current trace, and for all prefixes between that one and the current trace, φ holds. The *since* operator is the past dual of the future time *until* modality. The property $\ominus\varphi$ (previous-time φ) means that φ is true in a trace that is obtained from the current trace by omitting the last event. This is the past dual of the future time *next* modality. We can also define the following additional derived temporal operators: $\diamondminus \varphi = (\mathit{true} \ \mathcal{S} \ \varphi)$ ("past" or "once"), and $\boxminus\varphi = \neg \diamondminus \neg\varphi$ ("always in the past" or "historically").

Semantics A past time LTL formula is interpreted over a trace (or an observation) of events of the form $e_1e_2e_3\ldots$. Each event e is interpreted (labeled) with a finite set of propositions $L(e) \subseteq A$. This labeling is obtained when the event is observed. Let t_i denote the prefix trace $e_1e_2\ldots e_i$. The semantics of the logic is as follows:

- $t_i \models \mathit{true}$ is always true.
- $t_i \models q$ iff $q \in L(e_i)$.
- $t_i \models \neg\varphi$ iff it is not the case that $t_i \models \varphi$.
- $t_i \models (\varphi \ \mathit{op} \ \psi)$ iff $t_i \models \varphi$ op $t_i \models \psi$, when $\mathit{op} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.
- $t_i \models \ominus\varphi$ iff $i > 1$ and $t_{i-1} \models \varphi$.
- $t_i \models \diamondminus \varphi$ iff $t_j \models \varphi$ for some $1 \leq j \leq i$.
- $t_i \models \boxminus\varphi$ iff $t_j \models \varphi$ for all $1 \leq j \leq i$.
- $t_i \models (\varphi \ \mathcal{S} \ \psi)$ iff $t_j \models \psi$ for some $1 \leq j \leq i$ and $t_k \models \varphi$ for all $j < k \leq i$.

2.4 Runtime Verification of Past Time Linear Temporal Logic

Runtime verification is a technique used in software engineering to check that a system behaves correctly during its execution by monitoring and analyzing the events in its traces against specified requirements or properties. This process involves observing system executions in real-time or from recorded traces and using formal specifications to detect and possibly react to violations of expected behavior.

In [9], a runtime verification algorithm that takes a propositional past time linear temporal logic (LTL) formula and generates an efficient dynamic programming algorithm was proposed. The generated algorithm tests if a finite trace of events, provided as input, satisfies the formula and operates in linear time. The algorithm is based on calculating a *summary* for the current monitored trace. The summary is used, instead of storing and consulting the entire trace for providing verdicts, and is updated when new monitored events are reported. It consists of two vectors of bits. One vector, pre, keeps the Boolean (truth) value for each subformula, based on the trace observed so far *except* the last observed event. The other vector, now, keeps the Boolean value for each subformula based on that trace *including* the last event. Given a new event e which extends the monitored trace, the vector now is calculated based on the vector pre and the new event e. Every bit in now is associated with a different operator in the formula, each having a unique update pattern.

In RV, as in CTL, the processing of a formula commences after its subformulas are processed. Thus, for a $\psi = (\varphi_2 \wedge \varphi_2)$, we set $\mathsf{now}(\psi)$ to $\mathsf{now}(\varphi_1) \wedge \mathsf{now}(\varphi_2)$. For $\psi = \boxminus\varphi$, we set $\mathsf{now}(\psi) = \mathsf{now}(\varphi) \vee \mathsf{pre}(\boxminus\varphi)$. The full algorithm can be found in [9].

3 Our Methodology

Both model checking for CTL formulas and runtime verification for propositional past time LTL formulas can be handled with methods that share the same algorithm scheme. Our goal is to teach the LLM to produce a correct code for these two problems while providing only the minimal problem-specific guidance required to facilitate a solution.

Our methodology is a prompt-based technique. The first prompt describes the algorithm scheme, in this case *dynamic programming*, that is fundamental to both the runtime verification and model checking algorithms. Then, depending on the problem we wish to solve, we proceed with a second, succinct prompt that includes the formal specification of the verification problem as well as an instruction to follow the algorithm scheme to produce code that solves the desired problem.

We describe now the course of the algorithm scheme and its related data structures. The algorithm scheme outlines a general pattern for data propagation within a directed graph using designated *update functions*. In the two applications under focus, the nodes of such a graph represent states of the checked system, and an edge connects a state with its immediate successor. For every node in the directed graph, we hold a list of data items, where each data item has the following properties:

- *Data type* - may be any data type that is supported by programming languages.
- *Current value* - the value that is currently stored for the data item of a certain node.
- *Update function* - determines the update rule of the data item value.

An update function for a data item can modify its value based on the previous value, the values of other data items in the same node, or the values of data items from successor nodes in the graph. First, the data items of each node have their initial values, based on their update function. We then progress through each data item within every node, updating their values based on their associated update functions. This iteration continues repeatedly until we reach a point where all values remain unchanged. While convergence to final values for the data items is not guaranteed in the general case, for the two specific verification problems discussed, convergence is eventually achieved. Note that the specific programming examples in this study (Model Checking and Runtime Verification) are limited to data items with Boolean data types. However, other algorithms, such as those used in reinforcement learning, might employ different data types, such as numeric. The algorithm scheme prompt is presented in Listing 1.

```
Consider the following scheme for a data propagation
algorithm in a directed graph. The algorithm works on the
nodes of a directed graph. The graph has an initial state.
Every node in the graph includes some data items. Every item
has the following properties: update function, data type,
and value.
An update function of a data item can update its value based
on values of data items of the same node, or successor nodes.

The algorithm works as follows:
each data item of each node has an initial value. Then, we
iterate over all the nodes, and all the data items inside
the nodes, and update their values according to their update
values. We then iterate over all the data items again and
again, until the values of all the data items of all the
nodes are not changed.

This is only an algorithm scheme, in the next step you will
be provided with a use case that can be solved using this
scheme.
```

Listing 1: The prompt that was used to describe the algorithm scheme.

3.1 CTL Model Checking Algorithm Synthesis

Given the algorithm scheme, we aim to synthesize a CTL model checking algorithm that verifies a Kripke structure with atomic propositions attached to its nodes, with a minimum of additional information provided regarding the implementation details of such an algorithm. We devised a succinct prompt that consists of the following inputs:

- The task description - the LLM is instructed to use the scheme outlined in the previous prompt to generate a model checking algorithm for CTL formulas. Additionally, a description of the input requirements and the expected output is provided.
- The names of the temporal operators that are used in the relevant CTL formula.
- The CTL formula that represents the specification to be checked.
- The relation to the data propagation scheme - an explicit instruction to employ the scheme in creating the algorithm, accompanied by a subsequent request to represent subformulas of the CTL formula as data points in each node of the Kripke structure.

An example of such a prompt can be found on Listing 2. The algorithm developed from this prompt closely resembles the one described in [8]. We observed that, even in the absence of direct guidance, the LLM was able to grasp the data propagation scheme and utilized its intrinsic knowledge of formal verification and temporal logics to deduce the correct update functions for each of the CTL operators.

```
Based on the algorithm scheme provided, please generate a
model checking algorithm for CTL (Computation tree logic)
formulas. We want to model-check a directed graph, where
inside every node, every data point corresponds to a
subformula. consider only the logical operators not, and,
or, implication and double implication, and the temporal
operator AX (all-next).

Assume that the specification to be checked is "AX(q1) &&
AX(q2)" where q1 and q2 are boolean proposition. Please
describe the components of the boolean propagation algorithm
in this case. Then, generate a Python code for the
algorithm, where the main function (llm_model_check) gets a
directed graph as its only input. The code's output should
be a set of graph vertices that satisfy the specification.
```

Listing 2: An example of a short prompt that generates code that verifies a Kripke structure with respect to a CTL specification.

We provide here a code generation example of a model checking algorithm written in Python for the relatively simple specification $AX(q_0) \wedge AX(q_1)$, based on our methodology. In Listing 3, we show a part of the LLM response, where the different components of the model checking algorithm are described, with correspondence to the general algorithm scheme. We focus on the description of the update functions and the iterative update process. The response shows that the LLM understands how to apply the algorithm scheme for the structure of the temporal specification. Moreover, it successfully recognizes the recursive update function of the temporal operator AX.

```
Update Functions
Each data item needs an update function corresponding to its
logical or temporal nature:

q0 and q1: These are basic Boolean propositions and do not
require updates unless they are dynamically changing with
each iteration, which is typically not the case in static
model checking.

AX(q0) and AX(q1): These require temporal evaluation. The
value of AX(p), where p is a proposition, at a given node is
true if for all immediate successor nodes, the proposition p
is true. The update function here will look at all direct
successors of the node and check the truth of q0 and q1
respectively in those successors.

AX(q0) && AX(q1): This data item updates based on the values
of AX(q0) and AX(q1) at the node. It evaluates to true if
both AX(q0) and AX(q1) are true at the node.

Iterative Update Process
Propagation: Starting with the initial values for q0 and q1,
the algorithm iterates through all nodes to update AX(q0)
and AX(q1) based on the values of q0 and q1 in all successor
nodes.

Combination: After updating all nodes based on their
successors for AX(q0) and AX(q1), the values for AX(q0) &&
AX(q1) are updated.

Convergence Check: The process repeats (if necessary) until
there are no changes in the values across all nodes for any
data point.
```

Listing 3: The outline of the update functions and the iterative update process of a potential model checking algorithm for the formula $AX(q_0) \wedge AX(q_1)$, as described by the LLM. This desription is part of the LLM response to the prompt detailed in Listing 2.

The generated code of the model checking algorithm is depicted in Listing 4. The method *model_check* gets a Kripke structure as an argument, and outputs all the graph vertices that satisfy the specification. The code seems to precisely follow the guidelines of our algorithm scheme; a different update function is defined for each data item, and all the graph vertices and their data items are iterated until convergence. In the next section, we tested our approach against some non-trivial specifications.

```python
class Kripke:
    def __init__(self, R, L):
        # List of tuples (source, destination) representing directed edges
        self.R = R
        # Dictionary mapping nodes to sets of propositions (labels)
        self.L = L

def model_check(K):
    # Extracting nodes from edges and labels
    nodes = set([u for (u, v) in K.R] + [v for (u, v) in K.R] +
            list(K.L.keys()))

    # Create dictionaries to store AX(q0), AX(q1), and
    # AX(q0) && AX(q1) values
    ax_q0 = {node: True for node in nodes}
    ax_q1 = {node: True for node in nodes}
    result = {node: False for node in nodes}

    # Initialize AX values based on successor nodes
    for node in nodes:
        # Check all successors for q0 and q1 according to AX definition
        successors = [v for (u, v) in K.R if u == node]
        # If there are no successors, AX(q) is trivially False
        if not successors:
            ax_q0[node] = False
            ax_q1[node] = False
        else:
            ax_q0[node] = all('q0' in K.L.get(succ, set()) for succ in
                              successors)
            ax_q1[node] = all('q1' in K.L.get(succ, set()) for succ in
                              successors)

    # Main iterative loop to ensure convergence
    changed = True
    while changed:
        changed = False
        for node in nodes:
            new_ax_q0 = all('q0' in K.L.get(succ, set()) for succ in
                            successors)
            new_ax_q1 = all('q1' in K.L.get(succ, set()) for succ in
                            successors)
            new_result = new_ax_q0 and new_ax_q1

            # Check if any value has changed
            if new_ax_q0 != ax_q0[node] or new_ax_q1 != ax_q1[node] or
                           new_result != result[node]:
                ax_q0[node] = new_ax_q0
                ax_q1[node] = new_ax_q1
                result[node] = new_result
                changed = True

    # Collect nodes that satisfy the specification
    satisfying_nodes = {node for node, value in result.items() if value}
    return satisfying_nodes
```

Listing 4: The code of the model checking algorithm for the formula $AX(q_0) \wedge AX(q_1)$, as produced by the LLM.

3.2 Runtime Verification Monitor Synthesis

In [7], we proposed an LLM-based approach specifically for generating runtime verification monitors from natural language specifications. This approach consists of four stages, where the first one focuses on teaching an LLM to translate a given natural language specification to a past time LTL formula. Subsequent stages involve generating an RV monitor from the formula and verifying its correctness by submitting additional queries to the LLM.

In this work, we do not consider the translation stage from natural language but focus on synthesizing monitors from past time LTL formulas. The goal here is to leverage the information about the algorithm scheme and the knowledge the LLM possesses regarding runtime verification to generate a runtime verification monitor for a given specification. Relying on the algorithm scheme described in Listing 1 may allow a more succinct and easier description in the subsequent prompt.

We crafted a concise prompt that included the specification, the names of the operators used in the formula, and a general description of the algorithm purpose and setting, specifically that the object being verified is a trace of events. This prompt was designed to follow the algorithm scheme prompt from Listing 1. It was also necessary for the prompt to imply that the abstract syntax tree of the formula aligns with the graph outlined in the algorithm scheme. Without such an implication, the LLM had difficulty figuring out how to efficiently use the scheme to verify the trace against the specification. The *pre* and *now* boolean values of each syntax tree node are considered as two data items with Boolean data types. An instance of the presented prompt can be found on Listing 5.

```
Based on this scheme, create a runtime verification
algorithm for the following past time LTL formula: "H(q1 <->
P(P q1))". The "H" temporal operator means
"historically"/"always in the past", and "P" means "in the
last step"/"in the previous timestep".
The purpose of runtime verification is to get a trace of
events, where each event is an assignment for the boolean
variables, and output after every event if the trace
satisfies the formula so far (the verdict). In your
solution, the graph should be a tree that corresponds to the
syntax tree structure of the formula. Every operator has a
different update function. Clearly state the update function
for each operator. Remember that each update function may
rely on the previous value of the node to calculate its new
value.
```

Listing 5: An example of a short prompt that generates a runtime verification prompt for a given past time LTL formula.

During our experiments, we observed that the LLM under test effectively extracted the abstract syntax tree from the tested formulas. Following the classical dynamic programming algorithm for runtime verification suggested in [9], every past-time LTL operator has its own update rule. In our case, the update functions deduced for each operator were identical to the ones proposed in the paper. This highlights the high proficiency of the tested LLM in interpreting past time LTL operators and its ability to integrate them into an existing algorithmic pattern. This success is notably surprising, since past time LTL is not widely discussed in the literature. It was also interesting to see that the LLM identified that traversing the syntax tree nodes from bottom to top ensures that a single node traversal is sufficient for the boolean values to converge. Our experiments show that the algorithm developed from the prompt described in Listing 5 successfully verified traces against quite complex past time LTL formulas, with 20 or more syntax tree nodes.

We provide here a code generation example of an LLM-synthesized runtime verification monitor written in Python for the specification $\boxminus(q_1 \leftrightarrow \ominus(\ominus q_1))$, based on our methodology. In its output, the LLM first describes the past time LTL formula breakdown to a syntax tree and then determines the correct update function for every node. Subsequently, the monitor code is generated, as can be seen in Listing 6. The method *analyze_trace* gets a trace of events as an argument, and outputs a list of boolean verdicts, one for each observed event. As previously mentioned, the LLM realized that updating the values of the syntax tree nodes from bottom to top eliminates the need of executing the update statements for each node repeatedly. Therefore, the update statements for each syntax node are executed just once per event. The next section presents various experiments of this method applied to different specifications and traces.

```
def analyze_trace(trace):
    # Initialize the historical values
    historical_q1 = True
    previous_q1 = False
    previous_previous_q1 = False

    verdicts = []

    for event in trace:
        current_q1 = event['q1']

        # Update the 'P' nodes
        current_P_q1 = previous_q1
        current_PP_q1 = previous_previous_q1

        # Update the '<->' node
        current_biconditional = (current_q1 == current_PP_q1)

        # Update the 'H' node
        current_historical = historical_q1 and current_biconditional

        # Collect verdict for this event
        verdicts.append(current_historical)

        # Update historical values for next iteration
        historical_q1 = current_historical
        previous_previous_q1 = current_P_q1
        previous_q1 = current_q1

    return verdicts
```

Listing 6: An example of a runtime verification monitor code for the specification $H(q_1 \leftrightarrow P(Pq_1))$, as produced by the LLM.

4 Experimental Evaluation

We evaluated our approach for synthesizing both model checking programs and runtime verification monitors on specifications with varying sizes. The LLM under test was GPT-4, which was instructed to generate Python programs.

4.1 CTL Model Checking Algorithms Synthesis

We utilized our approach to generate model checking algorithms for six different CTL specifications of varying syntax tree sizes, where the smallest had five nodes and the largest had thirteen nodes. Each generated model checking program was compared to an existing Python-based model checking tool[1]. The comparison was made by testing their consistency across 1000 randomly generated Kripke structures with 10 to 30 nodes.

The model checking programs of five out of six of the tested specifications were completely consistent with the existing Python tool, where the model checker of the largest specification was consistent with only 401 out of 1000 of the tested Kripke structures. Again, GPT-4 accurately deduced the update functions for all the temporal operators. We attempted to generate some model checking programs to the same specifications without the algorithm scheme prompt, but could not get the recursive update functions for the different operators as obtained with the scheme.

4.2 Runtime Verification Monitors Synthesis

Initial Experiments We attempted to synthesize runtime verification monitors for nine propositional past time LTL specifications with varying syntax tree sizes. They were divided into three different groups based on their sizes, with three specifications for each group. Every generated monitor was manually inspected to ensure it produced the correct verdict. Additionally, every monitor was benchmarked against a real runtime verification tool to verify the consistency of the output verdicts across 1000 different traces, each containing 30 events.

Size (#nodes)	Correct Monitors	Correct Verdict Rate
4-6	3/3	100%
9-13	3/3	100%
16-20	2/3	94.3%

Table 1: Runtime verification monitor generation experiments. Each row represents a group of tested specifications of a different size.

Table 1 shows that our approach managed to generate correct monitors for eight out of nine of the tested specifications. It failed to generate a correct

[1] https://github.com/albertocasagrande/pyModelChecking

monitor for one specification from the last size group, with a syntax tree of 20 nodes. In this specification, only 82.9% of the inspected events yielded the correct verdict. We noticed that using our scheme, GPT-4 was able to deduce the update functions for all the temporal operators accurately. However, when the specification contained the *since* operator, we sometimes needed to emphasize that an update policy may use the previous values of its own node as part of the update function, otherwise a correct update rule for the since operator could not be inferred. Hence, for these specifications, we added the following sentence to the corresponding prompt: "remember that each update function may recursively rely on the previous value of the node itself and the current values of the operands of the node". In addition, we attempted to create the same monitors without providing the pattern provided in Listing 1, and only by using the concise, domain-specific prompt that is described in Section 3.2 but were not able to reproduce the results.

Comparison Against Other LLM-Based Monitor Generation Method In [7], we proposed an LLM-based approach specifically for generating runtime verification monitors from natural language specifications. To evaluate the approach, we devised 15 examples of natural language specifications that should finally be transformed into runtime verification monitors. The approach in [7] consists of four stages, where the first one translates the natural language specification to a past time LTL formula, and the rest of them generate an RV monitor out of the formula.

We evaluated both our current approach and the previous approach on the past-time LTL translations of the 15 natural language specification examples. Each past time LTL specification consisted of five to nine syntax tree nodes. When running the previous approach, we discarded the first stage and started from the second stage that gets past time LTL formulas as inputs. Again, every generated monitor of the two approaches was benchmarked against a real runtime verification tool to verify the consistency of the output verdicts across 1000 different traces, each containing 30 events.

Although the approach outlined in [7](excluding its first stage) employs seven different prompts compared to two prompts in the current work and features an overall prompt length about 3.7 times greater than ours, each method succeeded in generating correct monitors for 14 out of 15 examples. Our current approach not only shows competitive performance but also uses a generic programming pattern adaptable for code generation across various domains, unlike the prior approach that was specifically designed for RV monitors.

5 Conclusion

In this work, we present a methodology for synthesizing code through the use of LLMs. While LLMs are already widely used for code generation tasks, our approach focuses on achieving more reliable code with relatively shorter prompts.

The main principle in this work is to determine a programming pattern or scheme of our target code that is commonly used as a basis of algorithms and subsequently programs. In this paper, the scheme is that of dynamic programming [4]. We outline the programming scheme and the data structures employed to the LLM. As the LLM is likely to have been trained with many examples of this scheme, some verbal feedback is demanded, which increases our trust that the scheme is identified and will be correctly used. Following this, we introduce the specific algorithm for which we aim to generate code, detailing its main data structures. Lastly, we connect these data structures back to the scheme identified earlier and direct the LLM to produce the appropriate code for the algorithm.

We demonstrate the effectiveness of our approach by generating codes for both model checking and runtime verification algorithms that utilize the common underlying dynamic programming scheme. Our experiments show positive outcome, but it appears that existing LLMs find it challenging to process longer specifications for both algorithms. To evaluate our methodology, we attempted in further experiments to generate code for both algorithms using concise, domain-specific prompts, which omitted the description of the dynamic programming scheme. These experiments did not reproduce the results.

We believe that LLM-based programming methodologies such as the one presented here can be used to better harness LLM technologies for reliable code synthesis. Future research in this direction can explore the application of our approach with different programming patterns to generate complex code across various domains.

Acknowledgement

We thank Moran Omer for carefully reading and providing helpful comments on an early draft of this paper.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Comput. **2**(3), 117–126 (1987)
2. Anil, R., Borgeaud, S., Wu, Y., Alayrac, J., Yu, J., Soricut, R., Schalkwyk, J., Dai, A.M., Hauth, A., Millican, K., Silver, D., Petrov, S., Johnson, M., Antonoglou, I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T.P., Lazaridou, A., Firat, O., Molloy, J., Isard, M., Barham, P.R., Hennigan, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Doherty, R., Collins, E., Meyer, C., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Tucker, G., Piqueras, E., Krikun, M., Barr, I., Savinov, N., Danihelka, I., Roelofs, B., White, A., Andreassen, A., von Glehn, T., Yagati, L., Kazemi, M., Gonzalez, L., Khalman, M., Sygnowski, J., et al.: Gemini: A family of highly capable multimodal models. CoRR **abs/2312.11805** (2023)
3. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
4. Bellman, R.: Dynamic programming. science **153**(3731), 34–37 (1966)
5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on logic of programs. pp. 52–71. Springer (1981)

6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on logic of programs. pp. 52–71. Springer (1981)
7. Cohen, I., Peled, D.: End-to-end AI generated runtime verification from natural language specification. In: Bridging the Gap Between AI and Reality - First International Conference, AISoLA 2023, Crete, Greece, Submitted to the Post Proceedings volume (2024)
8. Grosu, R., Peled, D., Ramakrishnan, C., Smolka, S.A., Stoller, S.D., Yang, J.: Compositional branching-time measurements. In: From Programs to Systems. The Systems perspective in Computing: ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings. pp. 118–128. Springer (2014)
9. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.P., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 342–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
10. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 179–190. ACM Press (1989)
11. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D. (eds.) Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings. Lecture Notes in Computer Science, vol. 372, pp. 652–671. Springer (1989)
12. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: International Symposium on programming. pp. 337–351. Springer (1982)
13. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. OpenAI (2018)
14. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
15. Tan, C.W., Guo, S., Wong, M., Hang, C.N.: Copilot for xcode: Exploring AI-assisted programming by prompting cloud-based large language models. CoRR **abs/2307.14349** (2023)