End-to-End AI Generated Runtime Verification from Natural Language Specification*

Itay Cohen and Doron Peled

Bar Ilan University, Ramat Gan 52900, Israel

Abstract. We demonstrate how LLMs can be harnessed to synthesize runtime verification monitors from natural language specifications. We instruct the LLM through prompts to assemble the monitors through a sequence of stages. To start with, we employ the learned insights LLMs possess regarding linear temporal logic for translating natural language specifications into propositional past time LTL formulas. Next, we utilize a sequence of prompts to synthesize from the LTL specification a runtime verification monitor for the given formula. Part of these prompts enable the LLM to validate its own outputs, thereby significantly improving the probability of obtaining a correct monitor.

1 Introduction

Runtime verification (RV) [3–5, 10, 13, 16] allows monitoring the executions of a system either online or offline against a formal specification. The fact that each time a single execution is separately checked allows RV to be simpler than the more comprehensive model checking [8]. On the other hand, performing RV does not provide the level of correctness guaranteed by model checking. Still, RV plays an important role in the production of software systems; in particular, it allows shielding the system against malfunctions, and the collection of statistics about the executions of the system.

A recent trend uses LLM tools, such as ChatGPT and Gemini, for performing simple programming tasks. In fact, LLM-based tools, e.g., Copilot [19], aim specifically at lightweight synthesizing of code. While this kind of use for LLMs already appears on a small scale in the software development industry, the abilities of LLMs to provide useful code is considered to be somewhat limited. In essence, LLMs are trained to construct well-formed and informative sentences, responding to the requests that are prompted to them. This is achieved based on training appropriate neural networks using a huge number of text corpora. Among these texts, some programs and their corresponding descriptions are included, e.g., in repositories such as GitHub. However, LLMs typically do not include a serious deduction mechanism (as opposed to some shallow deduction that is inherent in the language mechanism). Such a mechanism is important

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

for the construction of complex code that includes intricate dependencies between objects and calculations based on nontrivial decision making. Another phenomenon that limits the usability of LLMs for synthesizing code is that they tend to be unreliable, sometimes providing an incorrect answer to a given prompt, rather than admitting the limitation in their mechanism to deliver the desired answer. This phenomenon is often called *hallucination*. This can, in our context, produce incorrect code. Thus, LLMs are particularly useful for the automatic programming of *simple* code, e.g., the interface between the internal representation of two systems, or simulating a finite state system based on its transition system.

We undertake here the task of teaching LLMs (specifically, GPT and Gemini) how to synthesize a runtime verification mechanism, given a natural language description of the requirement that needs to be checked or enforced. Thus, a natural language description is translated into code that performs runtime verification for the described property. This exploits the abilities of LLMs to capture and apply descriptions given in natural languages. It is important for our task, that the essential processing involved in the RV algorithm is simple enough, i.e., does not require the ability to apply deep deduction. This allows us to better trust the outcome of our construction. Furthermore, we apply various checks, also embedded within the LLMs and described via prompts, which enhance the reliability of the generated code.

A central principle that we adopt is to partition the task into smaller steps that sequentially follow each other. This includes (a) the construction of a translation mechanism from natural language to temporal logic, based on training examples, (b) the construction of an appropriate syntax tree that corresponds to the obtained temporal specification, (c) the construction of code that synthesizes the runtime verification engine and (d) the application of various testing procedures. All of these steps are performed within the LLM tools. This decomposition of tasks ensures that each task is simple enough to be enforced within the LLM. This helps us to overcome the limitations of current LLM technologies and harness their advantages. It allows exploiting the interfacing with natural languages, already embedded within the mechanism. Further, it employs simple steps, where the reliability of each one of them is independent of the ability of deep deduction. Our approach also exploits some checks that can be simply called for by prompting the LLM.

It is important to note that part of our process is based on instructing the LLMs, through prompts, to exploit, as part of the generated code, a classical RV algorithm [14]. Although the code for this algorithm is not *dictated* per se, our prompts clearly follow it; our process does not include a stage where this algorithm is somehow discovered by the LLM. It would be interesting to see if an alternative, more autonomous description, can be alternatively used, where the LLM itself discovers some connections, e.g., between the provided subformulas of the specification and the progress of information along the occurrence of the monitored events; essentially, this is the process of updating a *summary* of the execution that is needed with respect to monitoring the specification property.

The rest of the paper is organized as follows: Section 2 reviews LLMs and past time linear temporal logic, which is the temporal logic used for specifications in this paper. Each of the next sections describes a different stage in the process of synthesizing a runtime verification engine from natural language specification with the assistance of the LLM. Section 3 describes the various methods examined for translating a given natural language specification into past time LTL formulas. Section 4 delves into the process of converting these formulas into their abstract syntax tree representations through the use of LLMs. In this stage, we also employ validation prompts designed to increase the probability of obtaining a reliable output. Section 5 describes the stage where the output from the previous stage is used to construct a runtime verification monitor, using a single prompt that leverages the syntax tree based representation of the formula. In Section 6, we discuss the last stage of our pipeline, which involves using a version of the LLM that includes an interpreter component, to validate the monitor code synthesized in the previous stage.

2 Preliminaries

2.1 Propositional Past Time Linear Temporal Logic

Propositional past time linear temporal logic (PLTL) is a specification formalism that allows expressing safety properties [1]. 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:

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

The symbol q denotes a Boolean proposition over some finite set A of 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 the current trace that is obtained from the current one 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 $e_1 e_2 e_3 \ldots$. 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 suffix trace $e_i e_{i+1} e_{i+2} \ldots$. The semantics of the logic is as follows:

- $t_i \models true$ 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 $t_i \models \varphi$,
- $t_i \models (\varphi\ op\ \psi)$ iff $t_i \models \varphi$ op $t_i \models \psi$, when $op \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$,
- $t_i \models \ominus\varphi$ iff $i > 1$ and $t_{i-1} \models \varphi$,
- $t_i \models (\varphi\ \mathcal{S}\ \psi)$ iff $t_j \models \psi$ for some $1 \leq j \leq i$ and $t_k \models \varphi$ for all $j < k \leq i$.

2.2 Large Language Models and Linear Temporal Logic

Translating natural language to linear temporal logic has been studied before the deep learning era. These works mostly focused on handling structured sentences that were generated based on a certain grammar [7]. Recently, neural networks based methods have gained popularity over traditional approaches for translating into temporal logics.

Large language models (LLMs) have become a fundamental part of natural language processing (NLP), significantly enhancing the ability of computers to understand and mimic human language. These models, built on architectures such as *transformers* [21], have shown remarkable capabilities in various applications, from machine translation and question-answering systems to content generation and sentiment analysis. The essence of LLMs lies in their ability to process and generate text by learning patterns and relationships within vast datasets of human language. Their development involves training on extensive corpora, often encompassing billions of words sourced from books, articles, and the internet. This training enables the models to grasp the nuances of language, including grammar, idioms, and context. Notable LLM examples include OpenAI's GPT [18], Google's Gemini [2], and the more recent open-source models like Meta's LLaMA [20] and Mistral [15].

In [12], an LLM is fine-tuned based on a large dataset of LTL formulas. Other works focus on translating natural language specifications of a specific domain, such as grounded robotics [22]. Few-shot prompting techniques [6], in which we provide the model with input and output examples, also became prevalent in this domain. In [17], a sequence of prompts was used to translate natural language specifications for robots into an LTL specification, and [9] presented a framework that uses few-shot prompting to derive LTL formulas from unstructured natural language in an interactive way that involves human feedback.

3 Translating Natural Language Specifications into Past Time LTL Formulas

We present an approach for synthesizing a runtime verification mechanism given a natural language description of the requirement that needs to be checked. This general task is divided into four distinct sequential stages, each of which can be entirely executed using LLM tools. We conduct empirical experiments for each stage, to demonstrate its significance. This section outlines the first stage, with subsequent stages detailed in the following sections.

In this work, we consider natural language specifications that correspond to propositional past time LTL formulas. Such specification is appropriate for defining an agent that operates in an environment and can transition through different states or conditions while running. The majority of the examples tested involve agents like robots, which possess various internal states or conditions. An internal state may suggest, for example, that the agent is at a certain location (landmark) in the environment. An example of a natural language specification of this domain can be, for example, "the robot had to go to the repair station before entering sleep mode". In this example, the agent is a robot, and there is one landmark associated with it (the repair station), and one additional condition that may hold (entering sleep mode). Note that the specification domain is not restricted to the agent-environment setup described earlier. It can cover different areas, like specifications that describe properties software systems need to follow during an execution. However, our approach is versatile and can easily be modified to accommodate these variations.

At this stage, we specifically aim at translating natural language specifications into propositional past time LTL formulas. To do that, we decompose this stage into two subtasks, where in each subtask we harness the LLM's strengths via specific prompting techniques to increase the translation success rate:

- *Identification of landmarks and conditions.* An agent can be at a certain internal state (location), or condition, and this fact can be considered as a Boolean proposition. In this subtask, the LLM identifies these elements and replaces them with Boolean variables, similarly to the translation approach described in [17]. This results in a specification in an intermediate form, where there are variable placeholders instead of real propositions. An instance of an intermediate form specification could be: "whenever q_1 held and q_2 happened in the previous step, then one step before being at q_2 it was at q_3".
- *Intermediate form translation to past time LTL.* We once again employ the LLM to directly convert the specification in its intermediate form into a past time LTL formula, incorporating placeholders as Boolean variables.

3.1 Landmarks and Conditions Identification

We ask the LLM to identify locations or conditions of an agent that operates in an environment. Following the findings presented in [17], we also demonstrate high performance with GPT-4 on this subtask by providing a prompt with a clear description and a few examples (see Listing 1). The prompt, initially aligned with the scenario of an agent in an environment, can be revised to suit other natural language specification variants.

```
Your task is to repeat exact strings that refer to landmarks or conditions
of an object from a given utterance, and then replace each landmark or
condition with a different variable. the variables are q1, q2, q3 etc. After
identifying the landmarks or conditions, write the modified sentence inside
hashtags. You are provided with the following three examples:

1. utterance: the robot had to go to the repair station, and then to the
charging station.
Landmarks/conditions: the repair station; the charging station.
new sentence: # the robot had to go to q1, and then to q2. #

2. utterance: the vehicle should start at the garage, then go to the grocery
store and finally head back to the garage.
Landmarks/conditions: the garage; the grocery store.
new sentence: # the vehicle should start at q1, then go to q2 and finally
head back to q1. #

3. utterance: The phone must always be on silent mode, and eventually should
receive an MMS message.
Landmarks/conditions: be on silent mode; receive an MMS message.
new sentence: # The phone must always q1, and eventually should q2. #

Now, your task is to consider the following sentence:
"{insert_your_natural_language_spec_here}".
```

Listing 1: The prompt that was used for landmarks and conditions extraction. The input specification should be provided inside the curly brackets, and by using hashtags, the LLM is guided to deliver outputs that can be easily parsed.

3.2 Translating from Intermediate Form Specification into Past Time LTL

Many research efforts on temporal logic translations from natural language are focused on translations to LTL, with LLMs playing a key role in recent techniques. LLMs provide better results for subjects that were more common on their training dataset. Consequently, as LLMs have evolved, there has been a significant improvement in translation performance for formalisms such as LTL, which have extensive literature coverage. On the other hand, for formalisms that were less covered in the literature, like past time LTL, which is our selected intermediate specification formalism, the quality of these translations often falls short.

To demonstrate the limited knowledge of LLMs regarding past time LTL, we tasked GPT-4 to list the temporal operators of past time LTL and concisely define their semantics. In its response, the semantics proposed for some operators did not align with the standard semantics associated with past time LTL operators. More details on this are provided in Appendix A. To mitigate this, we experimented with two main approaches:

– *Fine-tuning based approach.* We created a dataset that consisted of 570 different pairs of sentences in natural language and their corresponding past time LTL formulas. Then, we fine-tuned GPT-3.5 on these pairs using a service provided by OpenAI[1], without introducing the model with past time

[1] https://platform.openai.com/docs/guides/fine-tuning

LTL. We developed our own dataset due to the absence of such resources, as past time LTL is a less common variant of LTL. We aimed to test the model's ability to generalize beyond the provided examples to those that it has never seen before.
Initially, we started with manually creating a smaller dataset of 190 different examples. After some preliminary experiments, we noticed that our fine-tuned model that is based on this dataset had a poor generalization ability; therefore we chose to augment it with the assistance of GPT-3.5. By asking it for two rephrasings of each natural language specification in the dataset, we were able to increase the dataset size threefold and introduce a greater linguistic diversity.

– *Prompting based approach.* Our goal here was to leverage the existing translation abilities of different LLMs, while introducing them to the concept of past time LTL using some prompting techniques. Similar approaches were considered in [9, 11, 17]. Initially, we devised a prompt detailing the various operators in past time LTL and their meanings. For certain operators, we highlighted their resemblance to future-time LTL operators, such as the connection between "always" and "historically". We then expanded the prompt by adding a second part, providing the LLM with three simple examples of translations (few-shot prompting). The full prompt is provided in Listing 2.

```
Your task is to translate the below natural language sentence into a
past-time LTL formula and explain your translation step by step. Remember
that P means "in the last step"/"in the previous timestep", S means "since",
H means "historically" and O means "in the past"/"once". The formula should
only contain atomic propositions or operators &&, ||, !, ->, <->, P, S, H,
O. The atomic propositions would be q1, q2, q3 etc. Imporatant note: do not
use the "globally" (G) operator from LTL, instead, use the H operator
("always in the past", "historically").

The "S" operator should be used in the following way:
natural language: "the robot was previously at q2, but since then it was
always at q1."
past-time LTL formula: " q1 S q2 ".

You are provided with three simple examples:
1. natural language: "The robot was at q1 at all times."
past-time LTL formula: # H(q1) #.
2. natural language: "Whenever the robot is was at q1, it is not at q2."
past-time LTL formula: # H(q1 -> !(q2)) #.
3. natural language: "In the past, the robot was at q3."
past-time LTL formula: # O(q3) #.

Write your final answer inside hashtags.

Please translate this sentence:
"{insert_your_intermediate_form_sentence_here}"
```

Listing 2: The few-shot prompt that was used for translating the intermediate form sentence to past time LTL. The sentence should be provided inside the curly brackets, and by using hashtags, the LLM is guided to deliver outputs that can be easily parsed.

Evaluation To evaluate the two approaches, we obtained a benchmark dataset with challenging examples. We based our dataset on an existing dataset that was used in [9]. In that work, five experts in the field were asked to provide pairs of natural language specifications and their formalizations into LTL. We observed that the majority of the examples in this dataset could be adjusted to reflect a past temporal perspective rather than a future one. Consequently, we manually altered 30 out of 36 natural language sentences to adopt a past form and paired them with their matching past time LTL translations. Note that the translation to a past form does not have to preserve the original semantics; it simply involves deriving one sentence from another. We provide below two examples of these modifications of temporal perspective in Table 1.

Original Sentence	**Sentence in Past Form**
q_1 never holds from some point in time on.	From the beginning of the execution until some point in the past, q_1 did not hold.
Whenever q_1 holds and q_2 holds in the next step, then q_3 holds one step after q_2.	Whenever q_1 held and q_2 happened in the previous step, then one step before being at q_2 it was at q_3.

Table 1: Two examples of the change in the temporal perspective

We started by evaluating our fine-tuning based approach for translating intermediate form specifications to past time LTL. Before evaluating it on the benchmark dataset, we explored its generalization abilities. We noted that the fine-tuned model often struggled to generalize to past time LTL patterns that were slightly different variants of the 570 examples in its dataset. For instance, the model encountered in its dataset the following sentences and their correct translations: "the robot was at q_5 three steps ago", "the robot visited q_4 two steps ago". However, it could not properly translate the sentence: "if the robot was at q_7 three steps ago, then it was at q_3 two steps ago". When we assessed its performance on our benchmark dataset, it reached a 30.0% accuracy rate.

We tested the prompting based approach on the benchmark dataset using four different configurations:

- *GPT-3.5-no-examples* - we equipped GPT-3.5 with the first part of the prompt that discusses the different temporal operators of past time LTL and their correspondence to regular LTL, without providing the few-shot translation examples at this point.
- *GPT-3.5-few-shot* - we used GPT-3.5 with the full prompt shown in Listing 2, including the three few-shot translation examples.
- *Gemini-few-shot* - we equipped Google's language model Gemini [2] with the full prompt.
- *GPT-4-few-shot* - we used GPT-4 with the full prompt.

During the experiments, we noticed that the configuration *GPT-3.5-no-examples* tends to mix the regular LTL operators with the past time LTL operators. It seems that it does not recognize the concept of "always in the past" and tends to confuse between "previous-time" and "past" operators. Moreover, the word "whenever" was not associated with the concept of the "historically" operator. Overall, this setup achieved an accuracy rate of 10.0% on the benchmark dataset.

After adding the three examples to the prompt of GPT-3.5 (*GPT-3.5-few-shot*), there was a noticeable improvement in its performance. However, the model still struggled to generalize in areas where it previously had difficulties. For instance, consider the natural language specification: "the robot is always at q_1, and whenever it is at q_3, it cannot be at q_2". The correct translation of this sentence is $(\boxminus q_1 \wedge \boxminus(q_3 \rightarrow \neg q_2))$, while the output translation was $(\boxminus q_1 \wedge (q_3 \rightarrow \neg q_2))$ (i.e., the *whenever* clause only applies to the first event). Overall, this setup achieved an accuracy rate of 43.33% on the benchmark dataset. The results were similar when we evaluated Gemini with the full prompt (*Gemini-few-shot*). In this case, the setup achieved an accuracy rate of 46.67%. Gemini seemed to misinterpret the notion of the "since" operator, and occasionally failed to differentiate between "previous-time" and "past".

The evaluation of the GPT-4 setup (*GPT-4-few-shot*) showed remarkable progress, outperforming all other configurations we examined. It successfully translated 28 of 30 sentences, achieving a 93.33% accuracy rate. It demonstrated a better grasp of the "since" operator and a more accurate use of the "previous-time" operator. The evaluation results of the two approaches and all the configurations are listed in Table 2. Due to its superior performance, we selected the *GPT-4-few-shot* configuration as the initial stage (translation stage) in the full monitor synthesis process.

Setup	GPT-3.5-ft	GPT-3.5-no-ex	GPT-3.5-fs	Gemini-fs	GPT-4-fs
Accuracy	30.00	10.00	43.33	46.67	93.33

Table 2: Evaluation accuracies (%) of all the translation approaches on the benchmark dataset.

4 Translating Past Time LTL Formula into Abstract Syntax Tree

The next stage towards synthesizing a propositional past time LTL monitor is teaching the LLM to analyze the formula that was generated in the previous stage. This analysis is conducted exclusively through prompts given to the LLM, specifically GPT-4 in this instance. We split this task into two subtasks, where each subtask is self-validated by the LLM, to increase the probability for a reliable output.

In the first subtask, we ask the LLM to create an abstract syntax tree (AST) out of the propositional past time LTL formula. In the prompt, as presented in Listing 3, we included all the possible operators and specified their arity. Aiming at a more standardized tree structure, we specified two further requirements: (1) leaves in the AST must only contain Boolean variables, without any operators, and (2) each AST node may have two immediate descendants at most. An example of a partial response for this prompt is shown in Listing 4. This subtask is followed by an additional prompt, presented in Listing 5, in which the LLM self-validates its previous output. In this validation, the LLM counts the number of leaves in the AST and checks if it is consistent with the number of Boolean variable occurrences. If this validation fails, we repeat this stage.

We experimented with two different approaches to repeat a subtask after failure. The first approach involved informing the model of its incorrect output and requesting an alternative solution. This method tended to yield solutions that were originally less likely to be produced and frequently incorrect. As a second approach, we removed the latest incorrect solution from the model message history, giving the model a fresh start as if we started a new conversation. This approach yielded better alternative solutions, so we applied it for all instances where repeating a task was required.

```
Past-time linear temporal logic is a language with q1,q2,...qn as boolean
variables. "!", "P","H","O" are unary operators, while "&&", "||", "->",
"<->", "S" are binary operators in this language.

Given a valid expression in this language, your task is to generate the
abstract syntax tree of it.
Do it for the following expression inside the backticks:
`{insert_past_time_ltl_formula_here}`.

Generate the abstract syntax tree while meeting the following requirements:
-do not attach unary nodes to the leaves.
-every node in the tree must have two children at most.
```

Listing 3: A prompt that creates a standartized AST out of a past time LTL formula. The formula should be provided inside the curly brackets.

```
### Abstract Syntax Tree Representation ###

          &&
         /  \
       H     H
      /       \
    q1        ->
             /   \
           q2    !
                  \
                  q3

This tree represents the structure of the expression `H(q1) && H(q2 -> !q3)`
according to the requirements:
- Unary operators (`H` and `!`) are not attached directly to leaves (`q1`,
`q2`, `q3`).
- Every node has at most two children, adhering to the binary structure of
the tree.
```

Listing 4: A partial response from GPT-4 to a prompt that asked for a standardized AST for the formula $H(q_1) \wedge H(q_2 \rightarrow \neg q_3)$.

```
Your next task is to count the number of leaves in the abstract syntax tree,
and the number of boolean variable occurrences in the formula provided
earlier.
Please write your final answer in the following format:

leaves count: <number>
variable occurrences count: <number>
```

Listing 5: A self-validation prompt that verifies that the number of leaves in the AST is consistent with the number of Boolean variable occurences in the formula.

In the second subtask, we take every non-leaf node in the AST and formulate an equation that outlines its relation with its immediate descendants, which are past time LTL subformulas. In addition, every leaf in the AST would be associated with its corresponding Boolean variable. Here, the LLM is required to assign an index to every node and generate the corresponding equations. The prompt for this subtask is described in Listing 6. For instance, the following equations should be generated for the formula $\diamondminus\, (q_1 \wedge q_2)$:

$$
\begin{aligned}
node_1 &= q_1 \\
node_2 &= q_2 \\
node_3 &= node_1 \wedge node_2 \\
node_4 &= \diamondminus\, node_3
\end{aligned}
$$

This subtask is also followed by a self-validation prompt that can be found in Listing 7. Here, the LLM is required to check if the number of operator appearances in the original formula is consistent with the number of equations that contain an operator. A failure in this test also leads us to initiate this stage again from the first subtask.

```
Your next task has two steps, given the same expression in this language:
1. Name every leaf and non-leaf node in the tree. The name format should be
"node{index}", where an ascending index would be in the placeholder. Start
from bottom to top. The first nodes should be the leaves. If two leaves
represent the same boolean variable, give them the same name.
2. Describe every node with the operator it represents and with one or two
of its immediate descendants. For example: "node2 = node3 && node6". If the
node is a leaf node, describe it as "node{index}=q1" if its corresponding
boolean variable is q1.

The order of the statements is important: you can use a certain node as an
operand in a statement only if its associated statement was already written.
You should finish with the root of the tree.
Please use the following format:
#####
<node description>
<node description>
...
#####
```

Listing 6: A prompt that creates a list of equations out of the AST of the past time LTL formula.

```
Your next task has two steps:
1. Count how many statements have an operator in their description. To do
that, thoroughly go over every statement and think if it has any operator
inside. Please explain your solution.

2. Count how many operators appearances are there in the original formula.
Remember, the different operators are: "!", "P", "H", "O", "&&", "||", "->",
"<->", "S". Please explain your solution.

Finally, make sure your answer is in the following format:
- operator statements: <number>
- operator appearances: <number>
```

Listing 7: A self-validation prompt that checks consistency between the number of operators appearances in the original formula and the number of equations that contain an operator.

Experiments We tested the approach of this stage on the benchmark dataset that we created, while focusing on the effectiveness of the self-validation checks of the two subtasks. For each past time LTL formula in the dataset, we generated its corresponding set of subformulas equations twice: initially omitting the self-validation checks and then including them. With the self-validation checks activated, the model was allowed up to ten attempts to correctly generate the equations. For these two variants, we report the quantity of formulas that were successfully handled, alongside those that failed during the syntax tree generation and equation generation steps. This experiment was conducted for both GPT-3.5 and GPT-4 models, with the findings detailed in Table 3.

Model	Approach	Success	Syntax Tree Fail.	Eq. Gen. Fail.
GPT-3.5	No Validation	11/30	11/30	8/30
	Self-Validation	23/30	3/30	4/30
GPT-4	No Validation	25/30	3/30	2/30
	Self-Validation	30/30	0/30	0/30

Table 3: Experimental results for the syntax tree generation stage on the benchmark dataset.

The table shows that self-validation of the LLM significantly improves the performance in this stage. The models produce probabilistic outputs, leading to potential inaccuracies concerning the subtasks under test. However, the two self-validation checks are designed to be simple, minimizing the chance of failure. Thus, passing these checks increases the probability that the stage's final output is accurate. Clearly, GPT-3's performance without self-validation is relatively poor, achieving a success rate of only 36.67%. Introducing self-validation boosts its accuracy to 76.67%. Given GPT-4's superior capabilities, the improvement is less pronounced, yet self-validation still enhances its performance.

5 AST Nodes to Runtime Verification Monitor

The third stage in our prompts pipeline is responsible for generating the monitoring algorithm for the desired specification. We employed a single GPT-4 prompt to produce a monitor coded in Python. In this work we consider propositional formulas, hence we assume that the input trace to the monitor consists of Boolean variable assignments.

Our prompt in this stage is a verbal outline of the classic monitoring algorithm presented in [14]. 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 consisting of a set of propositions, which extends the monitored trace, the vector now is calculated based on the vector pre and the event e. The symbol q denotes a single Boolean proposition. This is summarized below:

- $\mathsf{now}(\mathit{true}) = \mathit{True}$
- $\mathsf{now}(q) = (q \in e)$
- $\mathsf{now}(\neg\varphi) = \neg\mathsf{now}(\varphi)$
- $\mathsf{now}((\varphi\ \mathit{op}\ \psi)) = (\mathsf{now}(\varphi)\ \mathit{op}\ \mathsf{now}(\psi))$
- $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) = (\mathsf{now}(\psi) \vee (\mathsf{now}(\varphi) \wedge \mathsf{pre}((\varphi\ \mathcal{S}\ \psi))))$.
- $\mathsf{now}(\ominus\ \varphi) = \mathsf{pre}(\varphi)$
- $\mathsf{now}(\diamondminus\ \varphi) = (\mathsf{now}(\varphi) \vee \mathsf{pre}(\diamondminus\ \varphi))$
- $\mathsf{now}(\boxminus\ \varphi) = (\mathsf{now}(\varphi) \wedge \mathsf{pre}(\boxminus\ \varphi))$

When a new event appears, now becomes pre, and the now values are calculated according to the above cases.

The monitor is generated based on the prompt described in Listing 8 (the description of the classic monitoring algorithm), and on the equations that were generated in the previous stage, where each equation corresponds to a different subformula. Each node's value from the preceding stage corresponds to a single bit in the summary of the monitored trace.

Longer prompts that involve code generation with many requirements are known to be less reliable - as the prompt becomes longer, the likelihood of the code meeting all requirements decreases. Hence, we employed here an *adaptive prompting* technique. Most of the prompt is dedicated to the update rules for the now bit vector, although not every rule is needed in every case (as some operators might not be used in every formula). To reduce prompt length, we selectively include only the update rules that are relevant to the formula being addressed.

In this stage, we explicitly outlined the different update rules of each one of the relevant operators. We also began exploring alternative methods to apply these rules without directly mentioning them in the prompt, instead deriving them directly using the LLM. This early research direction is further described in Appendix B.

```
Your final task is to generate a Python program that receives at each step
an assignment for all the Boolean variables in the above Past-time linear
temporal logic expression, and then prints an output. This assignment of the
Boolean variables is referred to as an event. At each step, the event would
be inserted by the user. The output will be based on a certain analysis of
the event. The program would then be ready to receive another event from the
user and analyze it in a similar manner, and finally print another output,
and so forth. The program would terminate when the user inserted the word
"abort".
To perform the analysis, we would keep in memory two versions of the values
of each node from the previous task. The first version would be the current
version while the other one would be the previous version. Before the first
step, the current version of all nodes can be assumed to be a "False" value,
except of the current version of an "H" node that should be initialized to
"True".
The current version of each node will be updated based on its type and on
its previous version. The update order of the nodes should be the same as in
the previous task - the current version of the node may be updated only
after its operands are updated.
- A leaf node would be the truth value of the corresponding variable
according to the new event.
- The current version of node of type "!" would be the negation of the
current version of its single operand.
Similarly, we specify all the update rules that are relevant for the input
formula being addressed.
.....
.....
The previous version of all nodes will be updated only after the current
version of all nodes is updated. Finally, the current version of the root
node should be printed.
- When asking the user for an event, you must not print anything to the
user, just expect an input of the format "q1=<boolean>,q2=<boolean>,....",
where "boolean" is either "True" or "False".
- When printing the current version of the root node, print just a single
word that describes its boolean value (True or False).
```

Listing 8: A prompt that generates propositional past time LTL monitor, based on the subformulas equations and the monitoring algorithm description.

6 Self-Validation of the Monitor Code

6.1 Method

Code generated by LLMs based on prompts may be inaccurate on both syntactic and semantic levels. As part of our efforts to synthesize a runtime verification monitor using solely the capabilities of LLMs, we explored methods to validate the generated monitor. While we understand that a validation process started by LLMs might not be entirely accurate, our goal was to increase the likelihood of producing a reliable monitor. We aimed for two validation types: (1) syntactic validation of the monitor code, and (2) logical/semantic sanity check of the code.

We used GPT-4 to perform the self-validation stage. First, we queried it to produce two traces of events with respect to the original natural language specification (serving as a semantic sanity check), where one trace would satisfy the specification and the other would fail to do so. Then, we employed the code interpreter variant of GPT-4 to evaluate the monitor on the synthesized traces, and validate the output verdicts of the monitor (syntactic validation). We will now provide further details on these two steps.

Semantic sanity check with trace generation We noticed that GPT-4 is able to generate short traces that are consistent with past time LTL formulas and their natural language interpretation. We did some preliminary experiments in which we prompted GPT-4 with natural language specifications that correspond to past time LTL formulas, and asked it to output a short trace (up to ten events) that satisfies the specification. Each event was defined as an assignment to all the Boolean variables in the intermediate form sentence that was given as input. We then repeated this process but this time we provided the model with past time LTL formulas instead of their intermediate natural language representation. We observed that GPT-4 performs better when its input is a natural language sentence rather than a structured formula. We conclude that GPT-4 better captures temporal dependencies when the specification is provided in natural language.

Following this observation, we tested the trace generation ability on the benchmark dataset from Section 3.2. For each natural language specification in its intermediate form, we instructed GPT-4 to produce three short traces (up to ten events) that satisfy it, and three additional traces that do not. The prompt we provided for this task listed all the possible temporal operators of past time LTL, followed by their meaning. In addition, we provided a single-shot example triplet of a specification, a short satisfying trace and an unsatisfying trace, where each event was defined as an assignment to the different Boolean variables in the specification.

GPT-4 managed to output different traces for each specification, except two cases where there was a single satisfying trace. Among 90 traces designed to satisfy the specifications, 78 indeed satisfied their corresponding intermediate form sentences, marking an accuracy rate of 86.67%. Remarkably, it succeeded in producing accurate traces for the two samples that were not translated correctly in the first stage. For these samples, 5 out of 6 generated traces were accurate. Of the 90 traces that were not supposed to satisfy their specifications, 85 indeed did not, yielding an accuracy rate of 94.44%. The higher accuracy here may be attributed to the fact that, within our benchmark dataset, violating a few specifications tends to be slightly simpler than fulfilling them.

Syntactic validation with code-interpreter Recently, OpenAI has unveiled an improved version of GPT-4, namely the GPT-4 Code Interpreter[2]. This version is capable of offering reasoning in natural language while producing Python code in a step-by-step manner. Notably, it can generate and execute code incrementally, then relay the output of the executed code back to the LLM.

[2] https://platform.openai.com/docs/assistants/tools/code-interpreter

This variant of GPT-4 was utilized not only for syntactical verification of the generated code but also for executing it on two traces, one that satisfied the specification and one that did not, which were also produced with the help of the LLM. The prompt we provided had three requirements:

- Check that the monitor executes correctly, and that it produces a Boolean output for each event.
- Verify that the last verdict of the monitor for the satisfying trace is "True".
- Verify that the last verdict of the monitor for the unsatisfying trace is "False".

A failure in any of these checks requires us to go back to the code generation step and generate a new monitor. The prompt for this step is presented in Listing 9, and an example of a response is shown in Listing 10.

```
Given a first trace of events: {insert_satisfying_generated_trace}
and a second trace of events: {insert_unsatisfying_generated_trace}

Read the contents of the file "monitor.py", and modify it so instead of
getting inputs from the user, each time an event will be loaded from the
provided traces. When the first trace ends, rerun the code with the second
one. Make sure that each event is given as input in the format: "q1=True,
q2=False, ..".

Finally, let me know what were the outputs of the monitor for the two
traces. Summarize the process with three outputs:
1. Whether or not the monitor executed properly and produced a boolean
output for each event. (just write success/failure)
2. Write the last output that the monitor produced for the first trace.
3. Write the last output that the monitor produced for the second trace.

Write the summary in the following format:
execution status: (Success/Failure)
first trace last output: (True/False)
second trace last output: (True/False)
```

Listing 9: A prompt that uses the Code Interpreter variant of GPT to test the generated monitor on two traces. The traces to be tested should be provided inside the curly brackets, and the monitor itself is passed to the model as a separate file.

```
To address your request, I'll follow these steps:

1. Modify the code from "monitor.py" to process the provided traces instead
of reading from user input.
2. Load and run the first trace through the modified code.
3. Reload the code to start fresh and then run the second trace.
4. Summarize the results according to the format you provided.
First, I'll modify the code from "monitor.py" based on the description.

Let's start by writing the modified code to properly handle and iterate
through the two given traces.

.....
.....
<modified code displayed>
```

```
.....
.....

The modification of the code from "monitor.py" to process the provided
traces instead of getting inputs from a user was successful. Here's the
summary based on the results obtained:

execution status: Success
first trace last output: True
second trace last output: False
These outcomes indicate that the monitor executed properly for each event in
both traces and the specified final Boolean outputs for the last events were
True for the first trace and False for the second trace.
```

Listing 10: The response of the Code Interpreter variant after testing a generated monitor of two given traces.

6.2 Experiments

We conducted end-to-end experiments, applying all four stages of our proposed approach to synthesize runtime verification monitors from natural language specifications. The main goal was to evaluate the accuracy of the monitors produced, with an additional objective to test the effectiveness of the monitor self-validation technique described earlier.

Our experiments involved fifteen natural language specifications randomly selected from the benchmark dataset. In the first stage of the process, we used the *GPT-4-few-shot* configuration of the prompting based approach to translate from intermediate form specification into past time LTL. In the second stage, where the formula was translated into an abstract syntax tree representation, we allowed up to three repetitions in case of subtask failures. Subsequently, we permitted up to three attempts at generating monitor code (in the third stage) if there was a failure in the monitor self-validation stage (fourth stage). Once the process terminated, each monitor created was evaluated against a real runtime verification tool, regardless of the self-validation outcome. We based our comparison on 1000 distinct traces, randomly generated with 30 events each, where the comparative tool was supplied with the past time LTL formula derived from the respective natural language specification in the benchmark dataset.

Experiment	Correctness of Generated Monitor	Self-Validation Test Consistency
Accuracy	94.34	73.34

Table 4: Accuracies (%) of the end-to-end experiments.

Table 4 indicates that our four-stage process succeeded in generating accurate monitors for 94.34% of the specifications tested. Focusing on the self-validation test results, it was observed that these results were consistent with the actual correctness of the monitor in 73.34% of the cases (11 out of 15). Moreover, all discrepancies involved false negatives, where the self-validation stage incorrectly reported a correct monitor as faulty.

It is important to note that our work is still in progress, and the limited sample size in these experiments makes it challenging to draw definitive conclusions from the performance presented above. Nevertheless, these preliminary results are encouraging and represent a positive progression in the effort to teach LLMs the algorithmic foundations of runtime verification approaches.

7 Conclusion

In this work, we attempted to teach LLMs to synthesize code for runtime verification from natural language specification. While the LLM abilities were vital for the translation stage from natural language, we identified further opportunities to leverage natural language interfacing across additional stages of the synthesis process. Specifically, we employed a number of self-validation prompts that improved the creation of the syntax tree for a specific formula. Additionally, we observed that trace generation achieved greater accuracy when it is derived from the natural language specification, as opposed to the formula itself. This suggests that depending on the task domain, integrating language comprehension abilities can be beneficial at various points in the process.

Furthermore, at the current stage of LLM evolution, their inherent knowledge appears adequate for converting relatively complex natural language sentences into temporal logics, including less common variants such as past time LTL. For our purposes, drawing parallels between LTL and past time LTL, coupled with a few illustrative examples, sufficed to handle a variety of unstructured natural specifications. However, it is important to acknowledge that LLMs understanding of temporal dependencies is not flawless, and their translation efficiency declines as sentences increase in length and complexity.

For a future research, there is still work to be done with respect to teaching LLMs the algorithmic foundations of classic runtime verification approaches. In our work, the directives provided to the model in the stage of the runtime verification monitor creation were relatively straightforward. We anticipate that as LLMs continue to evolve, they will feature more advanced deduction capabilities, leading to a more profound comprehension of these algorithmic principles. Such advancements would, ideally, pave the way for instructing LLMs on comprehensive runtime verification concepts through more autonomous and concise explanations. We have already made a preliminary exploration in this direction by effectively extracting a few specific update rules for the RV summary using GPT-4. Further details on this are provided in Appendix B.

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. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. **20**(3), 675–706 (2010)
4. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1, https://doi.org/10.1007/978-3-319-75632-5_1
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444, https://doi.org/10.1145/2699444
6. 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. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
7. Brunello, A., Montanari, A., Reynolds, M.: Synthesis of LTL formulas from natural language texts: State of the art and research directions. In: Gamper, J., Pinchinat, S., Sciavicco, G. (eds.) 26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain. LIPIcs, vol. 147, pp. 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking, 1st Edition. MIT Press (2001), http://books.google.de/books?id=Nmc4wEaLXFEC
9. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 383–396. Springer (2023)
10. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005)
11. Fuggitti, F., Chakraborti, T.: NL2LTL - a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In: Williams, B., Chen, Y., Neville, J. (eds.) Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023. pp. 16428–16430. AAAI Press (2023)

12. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. CoRR **abs/2206.01962** (2022)
13. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods Syst. Des. **56**(1), 1–21 (2020). https://doi.org/10.1007/S10703-018-00327-4, `https://doi.org/10.1007/s10703-018-00327-4`
14. 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)
15. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., de Las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L.R., Lachaux, M., Stock, P., Scao, T.L., Lavril, T., Wang, T., Lacroix, T., Sayed, W.E.: Mistral 7b. CoRR **abs/2310.06825** (2023)
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009)
17. Liu, J.X., Yang, Z., Idrees, I., Liang, S., Schornstein, B., Tellex, S., Shah, A.: Grounding complex natural language commands for temporal tasks in unseen environments. In: Tan, J., Toussaint, M., Darvish, K. (eds.) Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA. Proceedings of Machine Learning Research, vol. 229, pp. 1084–1110. PMLR (2023)
18. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. OpenAI (2018)
19. 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)
20. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G.: Llama: Open and efficient foundation language models. CoRR **abs/2302.13971** (2023)
21. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA. pp. 5998–6008 (2017)
22. Wang, C., Ross, C., Kuo, Y., Katz, B., Barbu, A.: Learning a natural-language to LTL executable semantic parser for grounded robotics. In: Kober, J., Ramos, F., Tomlin, C.J. (eds.) 4th Conference on Robot Learning, CoRL 2020, 16-18 November 2020, Virtual Event / Cambridge, MA, USA. Proceedings of Machine Learning Research, vol. 155, pp. 1706–1718. PMLR (2020)

A GPT's Zero-Shot Understanding of Past Time LTL

When GPT-4 was instructed to list the temporal operators of past time LTL and concisely define their semantics, it proposed some operators that do not semantically align with the standard associated past time LTL operators. Specifically, the semantics of the proposed operator "until in the past" was not consistent with the conventional semantics of "since", which is the past dual of the future time "until" modality. In addition, it appeared not to recognize the "previous-time" operator. A prompt and a response that describe this phenomenon are shown in Listing 11.

```
User:
List all the temporal operators of past time linear temporal logic, and
shortly describe their semantics.

ChatGPT:
In linear temporal logic (LTL), temporal operators are used to reason about
sequences of events over time. Here are the primary temporal operators for
past time in LTL:

1. Globally in the past (P): Denoted as Pφ, this operator means that the
formula φ holds true for all past time points.
- Semantics: φ holds true at every point in the past.

2. Eventually in the past (P◊): Denoted as P◊φ, this operator means that φ
holds true at some point in the past.
- Semantics: There exists a past time point at which φ holds true.

3. Until in the past (PU): Denoted as PφUψ, this operator means that φ
holds true until ψ becomes true in the past.
- Semantics: For every point in the past where ψ holds true, φ holds true at
that point and all preceding points until ψ becomes true.

4. Release in the past (PR): Denoted as PφRψ, this operator means that φ
holds true until ψ becomes true, and once ψ becomes true, φ remains true for
all subsequent points in the past.
- Semantics: If ψ holds true at any point in the past, then φ has to hold
true at that point and all preceding points until ψ becomes true, and φ
continues to hold true for all past points thereafter.

These operators are used to reason about the past-time behavior of systems
and specifications in temporal logic.
```

Listing 11: GPT's zero-shot understanding of past time LTL.

B Extracting RV Summary Update Rules from GPT

We explored the possibility of extracting from GPT-4 the necessary update rules that help the classic RV algorithm update its summary, with only minimal information provided. We created a prompt that mentions the two Boolean vectors *now* and *pre*, the fact that each bit in the vectors is responsible for a different subformula, and the temporal operators "historically" and "since". Then, we requested it to formulate the update rules for these operators. GPT-4 was able to retrieve the correct update rules for these operators. as shown in Listing 12.

Note that the interaction detailed here represents the complete interaction with the model, with no preliminary knowledge provided.

```
User:
Your task is to build a runtime verification monitor that monitors
specifications written in propositional past time LTL. To track the boolean
values of each subformula along the execution, you have two Boolean vectors
- "now" and "pre" vectors. Assume that one bit of the "now" vector is
responsible for the formula "H(q)", where "H" is the "always in the past"
operator in past time LTL, and q is a subformula. how would you calculate
this bit, based on the other bits of the "now" vector and the bits of the
"pre" vector?

ChatGPT:
To build a runtime verification monitor that monitors specifications written
in propositional past time Linear Temporal Logic (LTL), you have to manage
the logical evaluation of LTL formulas over the traces of system execution.
The "H(q)" formula, where "H" denotes "always in the past" (historically),
requires that for the property "q" to be true at the current point in time,
it must have been true at all previous points in time during the execution.

Given two Boolean vectors, "now" and "pre", which track the truth values of
LTL subformulas at the current and previous points in time, respectively,
calculating the bit for "H(q)" involves the following steps:

1. Identify the Bit for q: First, identify the bit in both the "now" and
"pre" vectors that corresponds to the subformula "q". Let's denote these
bits as "now[q]" and "pre[q]", respectively.

2. Calculate H(q) for the Current Time: The truth value of "H(q)" at the
current time step depends on:
- The truth value of "q" at the current time step (i.e., "now[q]").
- The historical truth value of "H(q)" up to the previous time step (i.e.,
"pre[H(q)]").

3. Update Rule for H(q): The bit for "H(q)" in the "now" vector (denoted as
"now[H(q)]") can be calculated as follows:
- "now[H(q)]" = "now[q]" AND "pre[H(q)]"

This calculation is based on the semantic rule of the "H" operator, which
states that for "H(q)" to be true at the current moment, "q" must be true
now (i.e., "now[q]" must be true), and "H(q)" must have been true at the
previous moment (i.e., "pre[H(q)]" must be true). This reflects the property
that "q" must have always been true in the past for "H(q)" to hold.

User:
Now, how would you calculate the bit that corresponds to the formula "q1
Since q2"?

ChatGPT:
The "Since" operator in propositional past time Linear Temporal Logic (LTL)
captures a form of historical dependency between two conditions.
Specifically, the formula "q1 S q2" means "q1 has been true since q2 was
last true." In other words, for "q1 S q2" to be true at a certain point,
"q2" must have been true at some previous time, and "q1" must have been true
continuously from that point up to the current time.

Given two Boolean vectors, "now" and "pre", which represent the truth values
of subformulas at the current and previous time steps, respectively,
calculating the bit for "q1 S q2" involves considering both the current and
historical truth values of "q1" and "q2". Let's break down how to calculate
this bit:

1. Identify the Bits for q1 and q2: First, identify the bits in the "now"
and "pre" vectors corresponding to the subformulas "q1" and "q2". Let's
```

```
denote these bits as "now[q1]", "pre[q1]", "now[q2]", and "pre[q2]",
respectively.

2. Calculate "q1 S q2" for the Current Time: The truth value of "q1 S q2" at
the current time step depends on the current truth values of "q1" and "q2",
as well as the historical truth value of "q1 S q2". Specifically:
- If "q2" is true now ("now[q2]" = true), then "q1 S q2" becomes true
regardless of its past value because "q2" occurring now satisfies the
"since" condition.
- If "q2" is not true now ("now[q2]" = false), then for "q1 S q2" to be
true, both "q1" must be true now ("now[q1]" = true) and "q1 S q2" must have
been true in the previous step ("pre[q1 S q2]" = true).

3. Update Rule for "q1 S q2": Based on the above, the bit for "q1 S q2" in
the "now" vector (denoted as "now[q1 S q2]") can be calculated as follows:
- "now[q1 S q2]" = "now[q2]" OR ("now[q1]" AND "pre[q1 S q2]")

This calculation reflects the semantics of the "Since" operator: "q1 S q2"
is true if either "q2" is true now (capturing the idea that "q2" initiates
the truth of "q1 S q2") or if "q1" has been true since the last time "q2"
was true (reflected by "now[q1]" being true and "q1 S q2" also being true in
the previous step).
```

Listing 12: A prompt that extracts update rules of past time LTL temporal operators.