From Natural Language to Monitors via LLM-Assisted Disambiguation*

Itay Cohen[1], Klaus Havelund[2], Doron Peled[1], and Yoav Goldberg[1]

[1] Bar Ilan University, Ramat Gan 52900, Israel
[2] Jet Propulsion Laboratory, California Institute of Technology, USA

Abstract. Applying formal methods to a system typically involves expressing the requirements in terms of the native formalism of the used tool. In Runtime Verification (RV), this is often temporal logic. There is a potential gap between requirements expressed in Natural Language (NL), e.g., English, in which design and requirement documents are written, and their translation to the formalism. Our focus is the translation of descriptions written in NL to monitors. This poses several challenges. First, an NL description may be far distant from the basic constructs of the temporal logic, resulting in an unreliable translation. Second, the description may go beyond the expressiveness of the temporal logic. Finally, the description may include ambiguities that may result in a mismatch between the intended meaning and the formal specification. We present an LLM based method and an interactive tool named LLMon for synthesizing runtime monitors from natural language specifications. The tool supports two stages. In the first stage, it helps the user define, in natural language, a collection of monitorable temporal operators. For each operator, the tool assists the user in choosing among several possible interpretations of the initial natural-language formulation. The result is a monitoring procedure for each temporal operator. In this way, the user defines a custom temporal logic together with a corresponding monitor library. In the second stage, the user can again use natural language to define application-specific monitors as compositions of the monitors in the library. Because natural-language specifications can also be ambiguous here, the tool translates the resulting temporal specifications into automata and compares their languages, helping the user select the intended interpretation. The two stages help increase the reliability of the LLMon tool in synthesizing monitors from NL specifications.

* The research performed by Itay Cohen and Doron Peled was partially funded by Israeli Science Foundation grant 2454/23: "Validating and controlling software and hardware systems assisted by machine learning". The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

1 Introduction

We study the synthesis of monitors from Natural Language (NL) descriptions with the help of Large Language Models (LLMs). Such a monitor can follow the execution of a process, based on the captured events, and provide a verdict w.r.t. given requirements. It can also be used within a control loop, predicting some unwanted situations and averting them, or at least their escalation. This kind of monitor is often the focus of *Runtime Verification* (RV) [5–7, 19, 22]. The requirement is often written using a fixed formalism such as temporal logic or automata. RV tools typically synthesize monitoring code from such formal specifications.

In typical scenarios, users who wish to utilize runtime verification technology must be familiar with the native formalism supported by the tool that is used. Often, the constructs of such formalisms may not directly align precisely with the engineer's needs, and the user must therefore engage in a complicated *encoding*. This poses what we call the *"specification gap"*, common for many formal methods. In this work, we instead focus on the reliability of translating requirements formulated in natural language to formal specifications.

The natural language description may contain phrases that are far distant from the constructs of the formalism. This may result in unreliable translations[3]. Worse, some natural language descriptions may not even be expressible in the formalism. Finally, a natural language description may contain some ambiguity. Take, for example, the standard construct $a \mathcal{S} b$, which reads "a *since* b". One may assign different interpretations to it. The standard interpretation is that b happened in the past, and a happened ever since after that. However, it could be interpreted as a must have happened *at least once* after b happened. Another possible choice is whether the occurrence of b may refer to the most recent appearance, or alternatively the earliest one. All these interpretations are possible from just the word *since*.

In the first part of this work, we investigate a method and an interactive translation, based on using LLMs combined with RV synthesis algorithms such as [20]. Accordingly, we name our tool LLMon [1]. It can be used to (a) define and implement a monitoring logic in a step-wise manner, and (b) act as a corresponding monitor synthesizer from natural language requirements. This approach differs from similar works that employ LLMs in a more direct manner [11, 14, 17], translating requirements to an existing fixed temporal formalism. New past-time temporal operators and their corresponding sub-monitors can be defined and implemented through interactions with the tool. One can consider dozens of such operators, to mention only a few of them: since, previously, historically, previous-time, every k steps, value alternation, all except k times, stable-in-the-past, changes-only-once, initially, never, exactly-once, etc. With such an extended arsenal of operators, it now becomes more reliable to automatically translate from natural language descriptions to formulas. This would match natural language requirements more directly, and formulas can be more intuitive to read than specifications using only a small set of existing temporal

[3] The reliability of LLMs will increase over time, and such statements will have to be subject to re-evaluation.

modalities. This can be used by a programmer or a quality control engineer to build a monitor synthesis application that has a rich collection of constructs, which conform with the specification documents for a given (line of) product(s). Ultimately, this is a step in the direction of allowing RV users to write the specification in natural language, as close as possible to the given requirement in a design document.

Besides the advantages that LLMs have in automatically generating code, they are ideal for our task for the following reason: the ambiguity that appears in natural language, which is a main challenge in automatically translating natural language descriptions into a formal notation, is also inherent in LLMs, which are trained on human texts. This natural language ambiguity weakness, shared by both humans and LLMs, is turned into a strength by letting the LLM suggest different interpretations of a construct, which the user can then choose between.

We employ the following tactic for the selection of the correct interpretation for a new construct. When the user wants to introduce a new temporal construct together with its short natural language description, the LLM suggests several alternative interpretations for the construct, and generates code for each of these. Based on the user's preferences, the tool compares the verdicts provided by the alternative pieces of code that it synthesized on an exhaustive set of small sequences. Since this involves a single temporal construct, the set of sequences is typically limited to $3-4$ events. With distinguishing traces that demonstrate different verdicts for the distinct (re)formulation of the semantics for the new construct, the LLM explains to the user the difference between the definitions, thus assisting the user in choosing between alternative interpretations. One can consider this use of distinguishing sequences as *micro separation*.

Note that the user is *not* interacting directly with the LLM and does not directly write prompts that are fed to the LLM; prompts are generated by the implemented tool. These *internal prompts* also instruct the LLM to return responses in a format that can be processed by the tool, rather than as free text that is displayed to the user.

We limit the target specification logic to past-time constructs. as e.g., in past LTL. We allow user-defined constructs that are not directly expressive in this logic (e.g., a past version of Wolper's example in [31]). In fact, the tool can even synthesize operators whose semantics require unbounded memory, and thus go beyond first and second order monadic logic. However, our current operator-level ambiguity handling is designed for constructs with finite characteristics. We discuss the resulting limitations for such unbounded memory operators in the final section.

In the second part of this work, once the user-defined constructs have been acquired and integrated into the tool, we address the ambiguity that arises when translating a *complete* natural-language requirement into a temporal formula over this learned set of constructs. The LLM is invoked to produce such translations, yielding either a single formula or multiple candidate interpretations. When multiple candidates are produced, the tool generates distinguishing sequences that separate them by compiling the alternative formulas into automata and searching for (minimal) examples in the symmetric difference between each pair of candidates. These sequences are presented to the user, who

can use them to reject unwanted translations suggested by the LLM. This use of distinguishing sequences can be considered as *macro separation*.

Related work. The vast majority of works connecting LLMs and other deep learning models with linear temporal logic adopt the *direct* approach, where natural language requirements are translated directly into LTL. The topic has been explored even prior to the rise of deep learning. Earlier studies predominantly dealt with structured sentences parsed according to specific grammars [9, 13, 15, 24]. In recent years, the translation of natural language into temporal logics has been primarily driven by neural network-based methods.

In [17], the capability of fine-tuned language models to translate natural language requirements into formal specifications is investigated, specifically using the language model T5 [27]. Other works focused on using neural models for translating natural language specifications of a specific domain, such as robotics [25, 30]. Few-shot prompting techniques [8], which involve providing LLMs with input-output examples, have also become prevalent in this domain. The key idea is that, to translate natural language requirements into LTL formulas, the LLM is shown a few examples, each illustrating a different temporal operator, to convey their respective semantics. In [23], a sequence of few-shot prompts was used to translate natural language specifications for robots into an LTL specification.

The work in [11] presents a few-shot prompting system that derives LTL formulas from unstructured natural language by decomposing input sentences into sub-translations. Their work also addresses ambiguity in temporal natural language descriptions. The LLM maps fragments of the natural language specification to subformulas of the final formal output. For each fragment, it proposes several possible subformula translations, allowing the user to choose the most appropriate one or provide a custom translation through a menu of the translated options. In our work, the ambiguity is managed in a different way, which is assisted by the algorithmic generation of *distinguishing sequences*. This is done first at the construct level: the different interpretations suggested by the LLM are compared through code that exhaustively synthesizes short traces and compares their verdicts; the LLM then analyzes any distinguishing traces and explains the behavioral differences between interpretations, helping the user make an informed decision. Then, when the LLM is instructed to translate natural language specification into temporal formulas, we apply again an algorithmic approach to find distinguishing sequences between alternative translations. This is achieved through translating the obtained temporal formulas into finite automata and comparing their languages.

In the former version of LLMon, presented in [10], the tool focused primarily on construct acquisition: the user defines new temporal operators in natural language, the LLM proposes alternative operator-level interpretations together with update code, and ambiguity is resolved at the level of a single operator using short distinguishing traces. The current paper extends that framework to ambiguity in full specification translation. Here, the learned construct library becomes the target formalism for translating complete natural language requirements, and the translation stage is explicitly ambiguity-aware and may return multiple candidate formulas. Disambiguation is then carried out formally via

automata-based analysis, which generates distinguishing traces between candidate interpretations.

2 Preliminaries

Our method uses LLMs to facilitate synthesizing monitors for temporal specification. It combines the abilities of LLMs with RV techniques. Specifically, it generates a monitor in the style of the past-time LTL monitor in [20]. We describe here the standard logic and the size of the RV algorithm.

2.1 Propositional Past-Time Linear Temporal Logic

Propositional past-time linear temporal logic (PLTL) is a specification formalism that allows expressing safety properties [2]. The restriction to past time allows interpreting the formulas on finite traces.

Syntax. The formulas of PLTL are defined using the following grammar:

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

The symbol q denotes a Boolean proposition over some finite set A of *propositions*. 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 past (on a prefix of the execution observed so far) and φ has been holding since (on all prefixes between that one and the current trace). 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 operators: $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondsuit\, \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ ("past" or "once"), and $\boxminus\varphi = \neg \diamondsuit \neg\varphi$ ("always in the past" or "historically").

Semantics A past-time LTL formula is interpreted over a nonempty finite or infinite 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 $t_i \models \varphi$,
- $t_i \models (\varphi \ \vee \ \psi)$ iff $t_i \models \varphi$ or $t_i \models \psi$,
- $t_i \models \ominus\varphi$ 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 $j < k \leq i$.

In this work, we synthesize RV monitors for *past* temporal properties that extend the above past-time LTL fragment with additional (possibly user-defined) temporal operators.

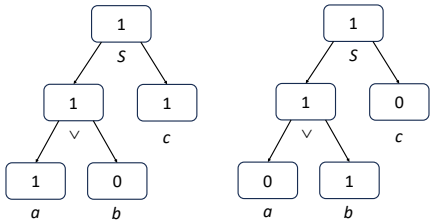

Fig. 1: Syntax tree and summary for $((a \vee b)\,\mathcal{S}\,c)$, given the trace $\langle a = 1,\, b = 0,\, c = 1\rangle; \langle a = 0,\, b = 1,\, c = 0\rangle$

2.2 The Synthesis Algorithm

The synthesis algorithm for past-time LTL translates the formula into a syntax tree. It uses a *summary* with one bit per each node of the tree, corresponding to a subformula. The summary keeps the truth value (0 or 1) of each subformula after any sequence of observed events. It is called a summary because it summarizes the information that the RV algorithm requires to keep after observing a trace; for past-time LTL, the summary is linear in the size of the specification and independent of the number of observed events in the trace. Upon intercepting a new event, the summary is updated. Figure 1 depicts the syntax tree and the summary generated for the formula $\varphi = ((a \vee b)\,\mathcal{S}\,c)$. The left tree is the summary after observing the event $\langle a = 1,\, b = 0,\, c = 1\rangle$, and the right tree is summary after further observing the event $\langle a = 0,\, b = 1,\, c = 0\rangle$. The update of the summary is according to the algorithm below, with η as its input formula. We denote by $\mathsf{pre}(\varphi)$ the summary value corresponding to the subformula φ before the update, and by $\mathsf{now}(\varphi)$ the value after the update. The order of updates is bottom up, according to the syntax tree.

$\mathsf{now}(\varphi) \leftarrow \mathit{false}$ **for each** subformula φ of η.
loop
  Observe a new event $s \subseteq A$ as input.
  $\mathsf{pre}(\varphi) \leftarrow \mathsf{now}(\varphi)$ **for each** subformula φ.
  for all subformulas φ of η **do**
   # If φ is a subformula of ψ then update $\mathsf{now}(\varphi)$ *before* $\mathsf{now}(\psi)$.
    $\mathsf{now}(\mathit{true}) \leftarrow \mathit{true}$
    $\mathsf{now}(p) \leftarrow (p \in s)$
    $\mathsf{now}(\varphi \vee \psi) \leftarrow \mathsf{now}(\varphi) \vee \mathsf{now}(\psi)$
    $\mathsf{now}(\neg\varphi) \leftarrow \neg\,\mathsf{now}(\varphi)$
    $\mathsf{now}(\varphi\,\mathcal{S}\,\psi) \leftarrow \mathsf{now}(\psi) \vee \big(\mathsf{now}(\varphi) \wedge \mathsf{pre}(\varphi\,\mathcal{S}\,\psi)\big)$
    $\mathsf{now}(\ominus\,\varphi) \leftarrow \mathsf{pre}(\varphi)$
  end for
  if $\mathsf{now}(\eta) = \mathit{false}$ **then report violation**; **exit**
  end if
end loop

One way to increase the expressiveness of past-time LTL is by adding to the temporal formula *transition rules* [18]. The rules use new propositions B, disjoint from the set of propositions A, and a set of rules of the form $b := L$, where $b \in B$ and L is a Boolean expression that can include propositions from $A \cup B$ and

the non-nested previous-time operator $\ominus$; when a proposition from B occurs, it must appear within the scope of the operator $\ominus$. For example, consider the rule

$$s := (c \vee (b \wedge \ominus s))$$

In fact, this example represents a redefinition for the *since* operator $\mathcal{S}$: s has the value of $(b \,\mathcal{S}\, c)$. Such rules can be translated directly into additional cases in the "for all" loop of the above algorithm. Variables within the scope of $\ominus$ are taken from the pre summary, and the others from the now summary. An additional bit, for the current value of s is added to the summary.

For instance, the size of the example by Wolper [31] shows that one cannot express that some (sub)property holds in every odd (even, depending also on how we count the index of the first event) state. But this property becomes expressive when adding the following simple rule to past-time LTL:

$$q := \neg \ominus q$$

Then, one can express that p holds in every odd state as $\boxminus(q \rightarrow p)$. Our approach allows adding constructs that make use of such update rules. It is shown in [18] that adding such rules increases the expressive power of past-time LTL to that of regular expressions or finite automata.

2.3 Large Language Models

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* [29], 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 training on vast datasets of human language. Their development involves training on extensive corpora, often encompassing a vast number of texts, sourced from books, articles, and the internet. LLMs are based on the principles of completing text by appending words. This allows unsupervised learning, since completing sentences and texts can be based on existing texts, a resource that is available in huge quantities over 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 [26], Google's Gemini [3], Anthropic's Claude [4], and xAI's Grok [32], as well as open-weight models such as Meta's Llama [28], Mistral [21], and DeepSeek [12].

3 An Overview of the Approach

We propose here a method that uses LLMs to assist in embedding additional temporal constructs into a tool that synthesizes RV monitors. The use of an LLM is ideal in bridging the gap between natural language and formal specification. Ambiguity is typical to natural languages, which makes it a challenge to translate natural language specifications into a monitor that has the intended

meaning. The same ambiguity is inherent in LLMs, through their training on natural language texts. We make use of their ability to reframe descriptions, associated in our case with temporal constructs, into alternative descriptions that reflect ambiguity, each accompanied by the corresponding monitor code for that construct.

Our method allows the user to introduce new temporal constructs to the specification formalism that can be used for synthesizing the monitor. We anticipate some possible ambiguity in the early definition of the construct, either inherent in the colloquial use of the used term in natural language, or because of some possible alternative interpretations that were unforeseen[4]. This is resolved in our approach by an interaction between the tool, the user and the LLM. Based on this interaction, the suggested tool generates prompts to the LLM; these prompts contain strict constraints, limiting the LLM responses to have a structure that can be processed by the tool. The user is never interacting directly with the LLM.

Extending the Classical RV Algorithm For each new temporal construct added to the specification formalism by the user, the tool synthesizes a Python function that implements an update rule in the style of the classical RV algorithm, as presented in the "for all" loop of the algorithm in Section 2.2. In the synthesized monitor, these functions will be combined together (including ones preprogrammed for the standard constructs such as *since* and *previous-time*). This seamless combination is partly facilitated by the compositionality property of the classical RV algorithm.

While the classical algorithm represents the monitor state as a one-bit truth value per subformula and uses the previously stored values pre to compute the next values during the bottom-up update (e.g., $\mathsf{pre}(\varphi \mathcal{S} \psi)$ for $\varphi \mathcal{S} \psi$ and $\mathsf{pre}(\varphi)$ for $\ominus\varphi$), our extension does not explicitly store prior Boolean values for all subformulas. Instead, each temporal operator occurrence is associated with a Python update function that is responsible for maintaining exactly the information it needs in order to compute its current truth value. This information is captured by a finite collection of *state variables* local to the operator, which are updated upon each new event and are not necessarily Boolean. In this sense, the notion of a "summary" becomes operator-specific: rather than being fixed to a single previous Boolean value, it is a flexible summary determined by the semantics of the construct. When the tool synthesizes an update function, it also synthesizes an appropriate initialization of these state variables, ensuring that each construct starts from the correct initial summary.

An RV monitor that follows this extended algorithm activates the update functions in a bottom-up order according to the formula's syntax tree. Upon observing a new event, suppose that a subformula ψ has already been updated and thus has an updated Boolean value for the current timestep. This value is then passed as an argument to the update function of ψ's parent operator, which is subsequently activated and updates its local state variables accordingly. These

[4] Giving graduate and undergraduate level exams that involve translating requirements into temporal logic is an excellent source of gathering alternative interpretations for such constructs, as manifested by students' questions during the exam.

state variables constitute the operator-specific summary, capturing exactly the information needed to compute the operator's next verdict when the next event arrives. We elaborate in the following sections on how the tool acquires such operators and composes their update functions into a complete RV monitor.

Listing 1 illustrates this principle on a standard past-time operator, *historically* ("always in the past"). In the classical one-bit summary view, the monitor would store the previous truth value of the subformula $\boxminus\psi$ and update it accordingly. In our setting, the operator is implemented directly as a Python update function whose state variable constitutes its local summary: the single field `truth_value` stores the finite memory needed to evaluate the operator on the observed prefix. Each time a new event arrives, the update function is invoked by the RV monitor, with the *current* truth value of its operand, passed as an argument. This operand is the truth value of the child subformula ψ after it has already been updated for the newly observed event in the bottom-up traversal. The function then updates `truth_value` to record whether a violation has occurred at any point so far. The required "past" information is encapsulated within this state variable, which is carried across updates and composed with the rest of the formula via the same bottom-up evaluation order.

```
class Historically:
    def __init__(self):
        self.truth_value = True

    def update(self, operand):
        if not operand:
            self.truth_value = False
        return self.truth_value
```

Listing 1: An example Python code that represents the update rule of the "historically" operator (always in the past).

High Level Workflow As the user introduces a new temporal construct with its utterance in English and a natural language description, the LLM is called to generate alternative texts ("reframing") that can describe the same construct or have a slightly different interpretation. The user can also add such alternative descriptions. The LLM is prompted by the tool to generate the code for the new update rules for these interpretations. Next, the user selects a primary interpretation from the available alternative descriptions - one that most closely aligns with their original intent. The code of the primary interpretation is then compared with the other interpretation codes in a pairwise fashion, by running each pair over an exhaustive set of traces. Since we compare such constructs in isolation with other constructs, it is often sufficient to generate traces of limited length to capture all the differences in behaviors.

The traces for which the generated update functions assign different verdicts are collected and displayed to the user. These are the *distinguishing traces*. Now, for each compared pair of codes, the tool provides the LLM with all the distinguishing traces found. The LLM gives a verbal description that explains why it is satisfied by one of the descriptions of the new construct and not by the other. These verbal descriptions are more intuitive than inspecting the distinguishing traces directly, enabling the user to understand how their primary interpretation

compares to the alternatives and ultimately make an informed selection about the desired version. This process of acquiring a new construct, which is the first stage of our approach, is also presented in Figure 2.

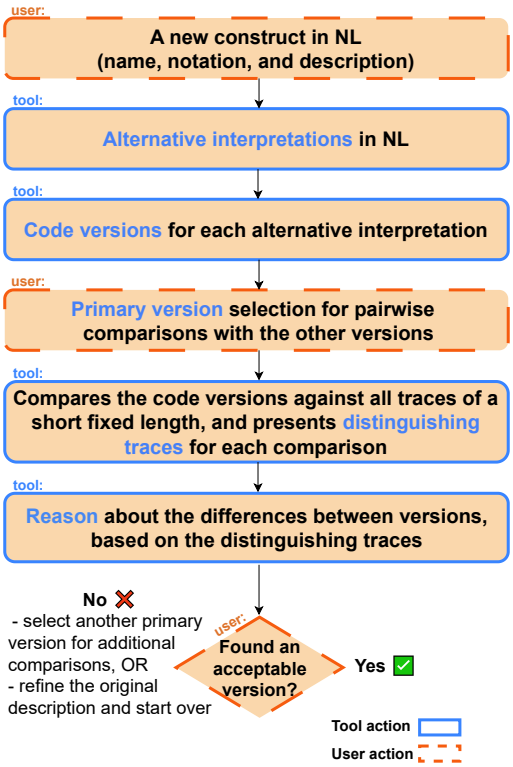

Fig. 2: A high-level flowchart of the formalism translation stage.

After the new constructs are acquired, the tool proceeds to the NL specification translation stage. Given a user-provided natural language specification and the resulting library of supported constructs (including the newly acquired ones), the tool prompts the LLM to assess whether the requirement is ambiguous with respect to the available constructs, and to produce either a single structured formula or a small set of candidate formulas. When multiple candidate translations are produced, the tool further generates distinguishing traces between them and presents these to the user (along with a visualization), enabling the user to select the intended interpretation.

Finally, once a structured formula is chosen (either provided directly by the user or selected from the translation candidates), the tool compiles the formula into an executable RV monitor.

In the next following sections, we illustrate the stages involved in both adding new constructs to the specification formalism, and the use of the tool for synthesizing RV monitors from natural language specifications. We do that using

a running example that showcases the interaction between a prospective user and the tool. During this interaction, the tool formulates prompts that are sent to the LLM without the direct intervention of the user. When presenting these prompts, we display only their core content, with full versions available in the Appendix.

4 Formalism Translation Stage

Assume the user intends to define a new construct that captures the fact that "the occurrence of one event appears *before* another". The tool asks the user to provide a name, a notation, and a natural language description for the construct. The construct information provided by the user does not need to follow a specific format, as it will be embedded in a prompt generated from their response and sent to the LLM, which can infer the user's intent from the free-form text. An interaction can look as follows.

Tool

Please provide a new temporal construct description (name, notation, natural-language semantics), and I will suggest some interpretations for it.

User

Construct name: "happened before".
Notation: q_1 HB q_2
Description: at any timestep, if q_2 holds, q_1 happened before that (or at the same timestep).

Based on this information, the tool assembles a prompt of the following form, which asks the LLM to create multiple (by default four) natural language interpretations ("reframings") that are slightly different from the original description. In addition, a Python code is requested by the prompt for every natural language interpretation. The full prompt is provided in Appendix, Listing 3.

```
Consider the following construct description: {user description}. Create
four different Python codes that calculate the truth value of the construct.
Each code should be based on a slightly different natural language
interpretation...
Important: the code should update the truth value of the operator in an
efficient way, using variables with finite domains.
```

The emphasis on efficiency and finite domain variables in the prompt leads the LLM to generate code that updates the construct truth value with minimal memory overhead, aligned with the classical RV algorithm. According to this prompt, the tool presents the LLM response. We show here only the first two generated interpretations and their associated code snippets. By default, the tool's user interface displays code snippets in a collapsed state, but users can expand them if they wish to view the code alongside the natural language interpretations. However, viewing the code may also be useful in some subtle cases, where ambiguity is not resolved using the different natural language descriptions.

Tool

Interpretation #1: q_1 can occur at the same timestep or before q_2 for the operator to be true.

```
    def __init__(self):
        self.q1_occurred = False
        self.operator_truth = True
    def update(self, q1, q2):
        if q2:
            if not self.q1_occurred and not q1:
                self.operator_truth = False
        if q1:
            self.q1_occurred = True
        return self.operator_truth
```

Interpretation #2: q_1 must occur strictly before q_2 for the operator to be true.

```
    def __init__(self):
        self.q1_occurred = False
        self.operator_truth = True
    def update(self, q1, q2):
        if q2:
            if not self.q1_occurred:
                self.operator_truth = False
        if q1:
            self.q1_occurred = True
        return self.operator_truth
```

At this point, the user needs to select a primary interpretation, the one whose natural language description most likely reflects their original intent. This interpretation will be compared to the others. Since interpretations are compared in pairs, choosing a primary interpretation reduces the number of comparisons and allows the user to focus on the most relevant ones. Note that the user can select as a primary interpretation a description that is not the one she suggested originally for the new construct. The user may also choose to ignore some of the provided versions at this step, or supply their own interpretation in addition to the existing ones, though the presented example does not cover that case.

User

Set interpretation #1 as the primary version for comparison.

Next, in a separate prompt (see the full version in Appendix, Listing 4), the LLM is instructed to generate code that compares the user-selected primary version with the alternative versions that remained. Each pairwise comparison should return all short traces of length k, where k is a user-configurable parameter set at execution time and should be at least 3, that may differentiate between the two implementations. Concretely, the generated comparison code exhaustively enumerates all possible traces of length k and simulates the update functions of the two versions along each trace. It then prints exactly the traces for which the resulting verdicts differ. This procedure is repeated independently for each pairwise comparison. A shortened version of the prompt is shown below.

```
Generate Python code that compares the primary version to all the other
versions, based on all the possible traces of length <k>...
```

Subsequently, the tool extracts the code from the LLM response, executes it, and presents the resulting distinguishing traces to the user. Traces are displayed using a compact visualization representing events as ellipses (inspired by [16]). Each ellipse represents a timestep (event), and the propositions are shown inside it, where a gray token indicates that the proposition does not hold at that timestep and a dark token indicates that it holds. For each candidate interpretation, the tool also indicates whether the shown trace is accepted or rejected. Only the first trace is shown by default; the remaining traces are collapsed (appearing in the tool's response in a line that starts with a ▸) and can be expanded by the user as needed. After observing this information, the user may proceed to the next step, where the tool generates a verbal comparison between the primary version and the alternatives based on the distinguishing traces.

Tool

Comparison: interpretation #1 vs. interpretation #2

▸ Expand for more traces

Comparison: interpretation #1 vs. interpretation #3
..

In the following step, the tool generates a prompt that directs the LLM to examine the given distinguishing traces for each pairwise comparison and identify the key differences between the versions. The prompt can be found in the Appendix, Listing 5. Note that the tool uses all the distinguishing traces generated per pair of interpretations (the primary one against the alternatives) to generate the text explaining the key differences. In the listing below, we show an example of such a generated text for a single pair of interpretations, based on the distinguishing trace presented above.

Tool

Trace analysis:
- Version 1: the operator is true even if q_1 occurs at the same timestep as q_2, allowing more flexibility in satisfying the condition.
- Version 2: the operator is false if q_2 occurs without q_1 having occurred strictly before it. This means that even if q_1 and q_2 occur simultaneously, the operator will be false.

Following this analysis, the user can either select a preferred version, choose a new primary version for additional comparisons, or revise their original description and restart the stage. Let us assume that the user selects Version 1.

User

Select version #1 as the implementation of the new construct.

Now, the tool prompts the LLM to return relevant details about the selected construct version. The relevant details include the chosen interpretation description, construct arity (unary or binary), construct symbol (notation), and the Python code implementing its update rule. To maintain compatibility with the upcoming monitor synthesis stage, the LLM is instructed to adhere to specific code design guidelines:

```
The construct class must define:
(1) An __init__ method that sets initial auxiliary variables, current
verdict and previous verdict.
(2) An update(arg) or update(arg1, arg2) method that updates and returns the
verdict. Make sure to actively update the verdict after every new event...

This function will have one argument for a unary operator and two for a
binary operator.
```

Once the LLM returns all these construct attributes, the new construct is added to a file containing a JSON array of all constructs generated by the tool. This array serves as the tool's construct knowledge base. An example of such a file is provided in the Appendix. At this point, the user may choose to define another construct or continue to the next stage.

5 NL Specification Translation Stage

5.1 Candidate formulas generation

In this stage, LLMon begins using the previously defined constructs as the target formalism for full specifications. It translates a user-provided natural language specification into a structured formula over these constructs, which is then used for monitor generation. To offer greater flexibility, the user may describe the specification either in natural language or directly as a structured formula. If a structured formula is provided, this stage is bypassed and the tool moves directly to monitor synthesis. Otherwise, the tool asks the LLM not only to translate the natural language specification, but also to assess whether it is ambiguous with respect to the available constructs. When ambiguity is detected, the output is a set of alternative candidate formulas rather than a single translation. In that case, LLMon additionally applies automata-based techniques to generate distinguishing traces that separate the candidate interpretations (i.e., traces that satisfy one candidate while violating another), helping the user understand the practical differences between translations before continuing to monitor synthesis.

LLMon uses a translation prompt to generate candidate formulas from the user's natural language specification. The notations and descriptions of the new constructs supported by the tool are presented to the LLM. Given that the new constructs are defined in natural language, the LLM can align fragments of the specification with these definitions and produce structured formulas using the corresponding constructs. However, as the number of custom constructs grows, multiple constructs may match the same fragment, yielding multiple reasonable translations. Therefore, the prompt instructs the LLM to explicitly assess whether the specification is ambiguous with respect to the available constructs. The LLM is expected to identify cases in which two or more constructs fit the

same part of the text and, when ambiguity is detected, to return all plausible formula interpretations (otherwise, it returns a single translation).

We demonstrate how the tool applies this translation prompt using the natural language specification shown below.

User

"Whenever a user writes to a file, it must have been created beforehand, and since file creation, the user has stayed logged in."

The tool appends the user specification, together with the JSON array describing the available constructs, to the translation prompt (see Appendix, Listing 7). The prompt guides the LLM through three steps:

1. Identify the events mentioned in the natural language specification and associate each event with a Boolean variable.
2. Assess whether the specification is ambiguous with respect to the available constructs, and if so, articulate the source of ambiguity.
3. Translate the specification into a structured formula over the available constructs and the Boolean variables; when ambiguity is detected, the LLM returns a set of alternative candidate formulas rather than a single translation. The prompt also includes a few NL-to-formula examples using standard past-time LTL operators, to illustrate how these operators should be applied in the translation and to reinforce the expected output notation.

Below we show the LLM output for the natural language specification entered by the user.

Tool

The following events were identified and associated with Boolean variables:
q_1 – file was created.
q_2 – user writes to a file.
q_3 – user is logged in.

The specification appears ambiguous. Possible translations:
1. $\boxminus((q_1 \text{ HB } q_2) \;\&\&\; (q_3 \text{ S } q_1))$
2. $\boxminus((q_1 \text{ HB } q_2) \;\&\&\; (q_3 \text{ WS } q_1))$

In this example, we assume that the tool's construct knowledge base includes not only the standard *since* operator (S), but also a *weak-since* operator (WS), whose semantics is similar but differs in the boundary case where the right-hand side never occurs. Intuitively, q_1 WS q_2 states that q_1 has been true, at least since the most recent time that q_2 was true; if q_2 was never true, then q_1 must hold for the entire observed sequence.

The tool flags the specification as ambiguous because the word *"since"* naturally suggests a past-time dependency, and in the tool's construct library it can be formalized either as S or as WS.

5.2 From formulas to monitor-automata

To support the distinguishing-traces analysis described next, each candidate structured formula φ is compiled into a deterministic finite automaton (DFA) A_φ that captures the monitor's finite-trace verdicts. The alphabet of A_φ consists of all possible truth assignments to the propositional events appearing in φ (e.g., $q_1, q_2, \ldots$). As in the classical monitoring algorithm, the tool maintains a *summary* over the syntax tree of φ. Each temporal operator occurrence is associated with a Python update function that maintains one or more finite-domain state variables; the current valuation of these variables constitutes the operator's *local summary*. For standard logical operators, there is no persistent memory to store; thus, their local summary is simply their current Boolean value under the present event.

DFA states. A DFA state represents the monitor's complete summary after reading a finite prefix. Concretely, it is a tree-shaped assignment of local summaries to the internal nodes of the parse tree of φ: for every temporal-operator node, the state stores a tuple containing the current values of its state variables (i.e., the values of the fields maintained by its update procedure), and for every Boolean-connective node, the state stores its current Boolean value. For negation, there is no need to store an explicit value, since it is always the Boolean complement of its operand. This representation is finite because in the first stage we enforce that all state variables range over finite domains. Therefore, only finitely many trees are reachable, and the resulting DFA is finite. A state of A_φ is accepting exactly when the local summary stored at the root indicates a positive verdict, i.e., when executing the root node's update yields True. In particular, the initial state is non-accepting: in the runtime verification setting, the empty trace carries no meaningful evidence, and we therefore represent it as a non-accepting state.

Automaton construction by state-space exploration. The tool constructs A_φ by exploring the reachable summaries induced by the different update functions. It starts from an initial state that records, for every temporal-operator node, the initial values of its state variables, and for every Boolean-connective node, an initial Boolean value. Then, for each discovered state and for each possible truth assignment to the propositional events of φ, the tool computes a successor state as follows. First, it loads the current state by assigning, for every temporal-operator node, the stored tuple of local-summary values back into the corresponding state variables of that operator update function. Next, it fixes the truth values of the propositional leaves according to the current assignment and evaluates the internal nodes bottom-up. At each temporal-operator node, the tool executes exactly one update step, providing as input the current truth values of the node's child(ren) under the same assignment; at each Boolean-connective node, it computes the connective's Boolean value from its child values. The successor state is obtained by collecting the updated state-variable tuples of all temporal operators together with the Boolean values of the Boolean-connective nodes. The transition relation is built by repeating this process via breadth-first search, adding newly discovered states until no new summaries are reachable.

5.3 Disambiguation using distinguishing traces

The tool now helps the user resolve the ambiguity between the natural language specification and its associated candidate formulas. This step is implemented via *pairwise comparisons*: for each pair of candidate formulas φ_1, φ_2, the tool aims to generate short finite traces that satisfy φ_1 but violate φ_2, and vice versa. The formulas φ_1 and φ_2 are constructed from the operators that LLMon has acquired, according to the first stage.

For a specification formula φ, denote the construction that returns its monitor-automaton (described in 5.2) by $\mathcal{A}(\varphi)$. To obtain a trace accepted by φ_1 but rejected by φ_2, the tool constructs an automaton for $\mathcal{A}(\varphi_1 \wedge \neg\varphi_2)$. This provides a simple way to generate the language difference of $L(\mathcal{A}(\varphi_1)) \setminus L(\mathcal{A}(\varphi_2))$, which avoids automata complementation and intersection (although for deterministic automata, complementation does not incur an exponential blow-up). Any accepting run of this automaton corresponds to a trace that distinguishes φ_1 from φ_2 in the desired direction. We find the shortest distinguishing trace by breadth-first search (BFS) over the automaton, returning the first accepting path found. The symmetric construction is applied to obtain a trace accepted by φ_2 but rejected by φ_1.

The resulting traces are then visualized and presented to the user, in the same style as done in the temporal operator translation stage. Based on these pairwise comparisons, the user selects the interpretation that best matches the intended meaning, and the tool proceeds with monitor synthesis for the chosen formula. We present below the tool's pairwise comparison of the two candidate translations produced by the LLM for the user-provided specification.

Tool

Comparison: Formula 1 vs. Formula 2.
- Every trace that satisfies formula 1 also satisfies formula 2.
- A trace that satisfies formula 2 and is rejected by formula 1:

In the non-ambiguous case, the tool presents the single inferred translation and asks the user to confirm it; if the user rejects it, they may revise and resubmit the natural language specification.

6 Monitor Synthesis Stage

Given a structured formula, the tool synthesizes a runtime-verification monitor by extending the classical bottom-up monitoring algorithm with user-defined constructs. A parser extracts from the tool's knowledge base the construct symbols and their arities (restricted to unary and binary operators) and uses them to parse the formula and build an abstract syntax tree (AST). For each temporal-operator occurrence in the AST, the synthesizer instantiates the corresponding Python update function and allocates its associated state variables, which constitute the operator's local summary. During monitoring, the tool updates the

AST bottom-up on each incoming event: propositional values are read from the event, Boolean connectives are evaluated directly, and temporal operators update their local summaries and return a truth value that is propagated to their parents. In this way, the monitor follows the structure of the classical algorithm while replacing the fixed one-bit pre summaries with operator-specific finite summaries encoded by the update procedures. The resulting monitor takes a list of events as input and returns a corresponding list of verdicts.

Note that once the tool obtains a structured formula for the specification, the LLM is no longer involved in synthesizing the monitor code. From this point on, parsing the formula, constructing the abstract syntax tree, and generating update statements for each construct are performed entirely by deterministic code. This design reflects our rationale to leverage the LLM only when it adds the most value, while avoiding its use in stages where hallucinations could introduce errors. In fact, in these stages, we used an algorithmic approach that helps in reducing the effect of hallucinations and increases the reliability of the translations.

The code generated from the formula $\boxminus((q_1 \text{ HB } q_2) \,\&\&\, (q_3 \text{ S } q_1))$ is presented below.

Tool

Your RV monitor (mon.py) was generated successfully!

```
# Temporal-operator helper classes (from constructs.json)
class HBOperator:
    def update(self, q1, q2): ...

class Since:
    def update(self, q1, q2): ...

class Historically:
    def update(self, q1): ...

def analyze_trace(trace):
    # instantiate temporal operators
    o_n3 = HBOperator()
    o_n6 = Since()
    o_n8 = Historically()
    verdicts = []
    for event in trace:
        v_n1 = event.get('q1', False)
        v_n2 = event.get('q2', False)
        v_n3 = o_n3.update(v_n1, v_n2)
        v_n4 = event.get('q3', False)
        v_n5 = event.get('q1', False)
        v_n6 = o_n6.update(v_n4, v_n5)
        v_n7 = v_n3 and v_n6
        v_n8 = o_n8.update(v_n7)
        verdicts.append(v_n8)
    return verdicts
```

7 Experiments

We used LLMon to define new temporal constructs based on natural language descriptions. We report here on experiments with thirteen constructs that were created, each with an associated name, notation and description. The first ten

constructs, shown in Table 1, have a corresponding form in past-time LTL, presented in the rightmost column of the table. The latter three are beyond the scope of past-time LTL, but can be expressed using the addition of transition rules [18], as shown in Table 2.

For all the constructs that we experimented with, their natural language description often reveals inherent ambiguity. We evaluated our tool using GPT-4o as the underlying LLM (via calls to OpenAI's API), setting the temperature to 0.1. Although we also tested with a temperature of 0, we found that introducing slight randomness encouraged more useful interpretation suggestions compared to fully deterministic outputs. While handling the construct descriptions with the tool, we aimed at finding interpretations that align with the intended semantics of their equivalent past-time LTL expressions. In the cases marked by '$*$' in Table 1, through using the interactions with the tool, our given initial description for a new construct turned out not to match our intended meaning of the new construct; this required us to provide an alternative description.

After obtaining the desired update rules for each construct, we aimed at verifying their correctness. We started with the verification of the first ten constructs that have past-time LTL equivalent representations. For each construct, we randomly generated one hundred structured formulas composed of standard logical connectives, the four standard past-time LTL temporal constructs, and the new construct (identified by its user-defined notation). The formulas varied in size, ranging from ten to twenty syntax tree nodes. We first used the tool's synthesizer component (introduced in Section 6) to generate RV monitors for each formula. Then, for each random formula, we created an equivalent formula by replacing any instance of a user-defined construct with its equivalent past-time LTL expression. We utilized the tool's synthesizer component again to generate an additional RV monitor, this time when the given input was the equivalent formula, which consisted only from the standard past-time LTL operators. We compared the two generated monitors using 10,000 randomly generated traces of 50 events each. In all the constructs tested, no discrepancies were detected in any of the generated formulas.

The last three constructs cannot be expressed as past-time LTL formulas. Consequently, we were unable to verify their correctness using the same approach applied to the other constructs. To properly verify these constructs, we had to be able to synthesize RV monitors that support *transition rules*, as presented in Section 2.2. We extended the tool's synthesizer component to support auxiliary Boolean variables, each defined by a transition rule. The extended synthesizer loads these variables and their rules from an external file. When one of these variables appears in the input specification, the synthesizer integrates the associated transition-rule-based update logic into the Python code of the monitor. We verified the correctness of each construct as done with the previously mentioned constructs, this time with the assistance of the extended synthesizer component. Again, no discrepancies were observed across any of the formulas.

Construct name	Notation	Natural language description	PLTL equivalence
weak since	q_1 WS q_2	q_1 has been true since the last time q_2 was true (one step after that). If q_2 never became true, q_1 must hold for the entire sequence.	$(q_1 \mathcal{S} q_2) \vee \boxminus q_1$
previous-time historically*	PTH(q_1)	q_1 was true from the beginning until one step before the most recent event.	$\ominus \boxminus q_1$
value alternation	VA(q_1)	q_1 should change its value at every step.	$\boxminus \left(\ominus true \rightarrow (q_1 \leftrightarrow \ominus(\neg q_1)) \right)$
consistent Boolean value	CBV(q_1)	true if q_1 is the same across all timesteps.	$\boxminus q_1 \vee \boxminus(\neg q_1)$
never	N(q_1)	q_1 never holds.	$\boxminus(\neg q_1)$
happened before*	q_1 HB q_2	at any given timestep, it is true that if q_2 holds, q_1 happened before that (or at the same timestep).	$\boxminus (q_2 \rightarrow \diamondminus q_1)$
false since*	q_1 FS q_2	since one step after the latest occurrence of q_2, q_1 has been false. If q_2 was never true, q_1 FS q_2 is true.	$\diamondminus q_2 \rightarrow (\neg q_1 \mathcal{S} q_2)$
false before*	q_1 FB q_2	Before the current q_2, q_1 was always false.	$q_2 \rightarrow \boxminus(\neg q_1)$
at least once from start	OAS(q_1)	q_1 has been true at least once from start.	$\diamondminus \left(\boxminus q_1 \right)$
changes only once	COO(q_1)	q_1 changes its value once in the sequence.	$(q_1 \wedge (q_1 \mathcal{S} \boxminus \neg q_1)) \vee (\neg q_1 \wedge (\neg q_1 \mathcal{S} \boxminus q_1))$

Table 1: Temporal constructs with past-time LTL equivalent expressions.

Construct name	Notation	NL description	Auxiliary variables	Equivalent formula
holds at odd	ODD(q_1)	q_1 holds at odd timesteps.	$a_1 := \neg \ominus a_1$	$\boxminus(a_1 \rightarrow q_1)$
holds at some divisible by 3	DIV3(q_1)	q_1 holds at some timestep divisible by 3.	$a_1 := \neg(\ominus^2(true) \rightarrow (\ominus^3(true) \wedge \neg \ominus^3 (a_1)))$	$\diamondminus (a_1 \wedge q_1)$
consistent odd and once even	COOE(q_1)	q_1 has the same value at all odd timesteps, and holds in at least one even timestep.	$a_1 := \neg \ominus a_1$; $a_2 := \neg(\ominus(true) \rightarrow \ominus a_1)$	$(\boxminus(a_1 \rightarrow q_1) \vee \boxminus (a_1 \rightarrow \neg q_1)) \wedge (\diamondminus (a_2 \wedge q_1))$

Table 2: Temporal constructs beyond the scope of past-time LTL. ($\ominus true$ means that there *is* a previous state)

8 Unbounded Memory Constructs

In the previous sections, we have focused on user defined past-time constructs whose monitoring semantics can be realized with *finite* summaries. This includes constructs expressible in past-time LTL, as well as constructs that go beyond it but can still be encoded via the addition of transition rules. In these cases, LLMon can enforce that all operator state variables range over finite domains, which enables (i) operator-level ambiguity resolution via short distinguishing traces, and (ii) specification-level disambiguation, where alternative candidate formulas are separated using automata-based analysis, as described in the previous sections.

A different class of constructs relies on *unbounded* memory. Such operators cannot, in general, be represented by a deterministic finite automaton, and specifications that combine them go beyond the bounds of first and second-order monadic logic. Importantly, LLMon is still capable of synthesizing update functions for such operators from natural language descriptions. However, the finite-state disambiguation mechanisms no longer apply in general.

Example: a FIFO-style operator. Listing 2 shows an operator that maintains an integer counter `buffer_size` and rejects whenever a "pop" event occurs while the counter is zero. Intuitively, if q_1 denotes a "push" operation and q_2 denotes a "pop" operation, the operator enforces the safety constraint that at every point in the execution prefix, the number of observed q_2 events does not exceed the number of observed q_1 events. This requires unbounded memory in general: `buffer_size` can grow arbitrarily with the trace length, so the operator's local summary cannot be captured by a finite number of bits.

```
class FIFOOperator:
    def __init__(self):
        self.buffer_size = 0
        self.is_valid = True

    def update(self, q1, q2):
        if q1:
            self.buffer_size += 1

        if q2:
            if self.buffer_size == 0:
                self.is_valid = False
            else:
                self.buffer_size -= 1

        return self.is_valid
```

Listing 2: An example of a user defined construct whose monitoring semantics requires unbounded memory.

Limitations in ambiguity handling. For unbounded memory operators, the tool can still generate plausible code implementations from natural language, but its current ambiguity handling becomes inherently limited. First, operator level disambiguation based on short distinguishing traces may fail to expose differences that manifest only on longer prefixes. Thus, selecting between alternative interpretations becomes an approximation, and correctness cannot be guaranteed solely from short counterexamples. Second, the specification level disambiguation stage is not applicable because it relies on compiling candidate formulas into

finite deterministic automata, which in turn requires that each operator's state variables range over finite domains. Consequently, when unbounded memory operators are allowed, the tool may still translate and synthesize monitors, but the automata-based distinguishing traces analysis used to separate alternative specification translations is not feasible.

One may ask whether the automata-based analysis used in the bounded-memory setting can be extended by compiling formulas with unbounded-memory operators into pushdown automata. Even if we assume that two candidate formulas can individually be represented by a pushdown automaton, generating distinguishing traces between them requires reasoning about their symmetric difference. This, in turn, reduces to checking non-emptiness (and, when non-empty, extracting a short witness) for the intersection of two pushdown automata. However, these non-emptiness and witness-finding tasks are undecidable in general for such intersections, which prevents a general automata-based distinguishing-trace procedure in the unbounded-memory setting.

Small-scale experiments. Despite these limitations, we experimented with a few unbounded memory constructs in a small-scale setting, including `equals-to`, `greater-than`, and `FIFO` (Table 3). To enable their generation, we removed the finite domain variables restriction from the prompt described in Listing 3 (provided in the Appendix). Since we did not have a reference monitor library for these constructs, we validated them by manually designing, for each operator, two representative formulas together with traces that are known to satisfy and violate each formula. We then synthesized monitors for these formulas using the tool and evaluated them on these traces. Across these tests, the monitors accepted and rejected exactly as expected, and no discrepancies were found.

Construct name	Notation	NL description
greater than	q_1 GT q_2	The number of occurrences of q_1 is greater than or equal to the number of occurrences of q_2 at this point.
equals-to	q_1 EQTO q_2	The number of occurrences of q_1 is equal to the number of occurrences of q_2 at this point.
FIFO	q_1 FIFO q_2	q_2 never occurs when the buffer is empty, where each q_1 adds an item and each q_2 removes an item.

Table 3: Examples of unbounded-memory constructs explored in our experiments.

9 Conclusions

This work explored a step-wise, LLM-assisted workflow for building runtime verification monitors directly from natural language requirements. We presented our approach that involves incremental human-in-the-loop construction that combines LLM guidance with formal, trace-based disambiguation. The results show

that the LLM can help to refine and implement new temporal constructs whose semantics more closely match the intent expressed in natural language, and extend the expressiveness of the underlying logic, without forcing the user to master formal syntax.

By asking the LLM to propose multiple candidate interpretations, the workflow surfaces ambiguity rather than hiding it. In fact, ambiguity is resolved twice: first at the *operator level*, where alternative meanings of a newly introduced construct are validated and compared using distinguishing traces, and later at the *specification level*, where complete natural language requirements may admit multiple formalizations over the learned constructs. In both cases, automatically generated distinguishing traces provide concrete evidence that helps the user reject unintended interpretations and select the intended one, turning ambiguity into a mechanism for surfacing subtle design alternatives that might otherwise be overlooked. Note that all safety-critical tasks - formula parsing, monitor composition, and large-scale trace evaluation - remain deterministic. This division of labour minimizes the risk of LLM hallucinations by keeping humans "in the loop" for every semantic decision, and consequently allowing more reliable translation of natural language requirements to monitors.

For further work, we want to extend the method presented in this paper to first-order specifications that allow events with data [7, 19].

References

1. `LLMon` tool source code. `https://github.com/itay99988/LLMon` (2025)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Comput. **2**(3), 117–126 (1987)
3. Anil, R., Borgeaud, S., Wu, Y., Alayrac, J., et al.: Gemini: A family of highly capable multimodal models. CoRR **abs/2312.11805** (2023)
4. Anthropic: Claude 3.7 sonnet system card (Feb 2025), `https://www.anthropic.com/claude-3-7-sonnet-system-card`
5. 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)
6. 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)
7. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
8. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., et al. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
9. 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., et al. (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)

10. Cohen, I., Havelund, K., Peled, D., Goldberg, Y.: The power of reframing: Using llms in synthesizing RV monitors. In: Könighofer, B., Torfah, H. (eds.) Runtime Verification - 25th International Conference, RV 2025, Graz, Austria, September 15-19, 2025, Proceedings. Lecture Notes in Computer Science, vol. 16087, pp. 202–212. Springer (2025)
11. 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)
12. DeepSeek-AI: Deepseek-v3 technical report. CoRR **abs/2412.19437** (2024)
13. Finucane, C., Jing, G., Kress-Gazit, H.: Ltlmop: Experimenting with language, temporal logic and robot control. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (2010)
14. 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., et al. (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)
15. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: Sabetzadeh, M., Vogelsang, A., Abualhaija, S., et al. (eds.) Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org (2020)
16. Greenman, B., Saarinen, S., Nelson, T., Krishnamurthi, S.: Little tricky logic: Misconceptions in the understanding of LTL. Art Sci. Eng. Program. **7**(2) (2023). https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/7, https://doi.org/10.22152/programming-journal.org/2023/7/7
17. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. CoRR **abs/2206.01962** (2022)
18. Havelund, K., Peled, D.: An extension of first-order LTL with rules with application to runtime verification. Int. J. Softw. Tools Technol. Transf. **23**(4), 547–563 (2021)
19. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods Syst. Des. **56**(1), 1–21 (2020)
20. 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)
21. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., et al.: Mistral 7b. CoRR **abs/2310.06825** (2023)
22. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009)
23. Liu, J.X., Yang, Z., et al.: Lang2ltl: Translating natural language commands to temporal specification with large language models. In: Workshop on Language and Robotics at CoRL 2022 (2022)
24. Nikora, A.P., Balcom, G.: Automated identification of ltl patterns in natural language requirements. In: 2009 20th International Symposium on Software Reliability Engineering (2009)

25. Oh, Y., Patel, R., Nguyen, T., Huang, B., Pavlick, E., Tellex, S.: Planning with state abstractions for non-markovian task specifications. In: Bicchi, A., Kress-Gazit, H., Hutchinson, S. (eds.) Robotics: Science and Systems XV, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019 (2019)
26. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. OpenAI (2018)
27. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., et al.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research (2020)
28. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., et al.: Llama: Open and efficient foundation language models. CoRR **abs/2302.13971** (2023)
29. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., et al.: Attention is all you need. In: Guyon, I., von Luxburg, U., et al. (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)
30. 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., et al. (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)
31. Wolper, P.: Temporal logic can be more expressive. In: 22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981. pp. 340–348. IEEE Computer Society (1981)
32. xAI: Grok 4 model card (Aug 2025), `https://data.x.ai/2025-08-20-grok-4-model-card.pdf`

A The user interface of LLMon

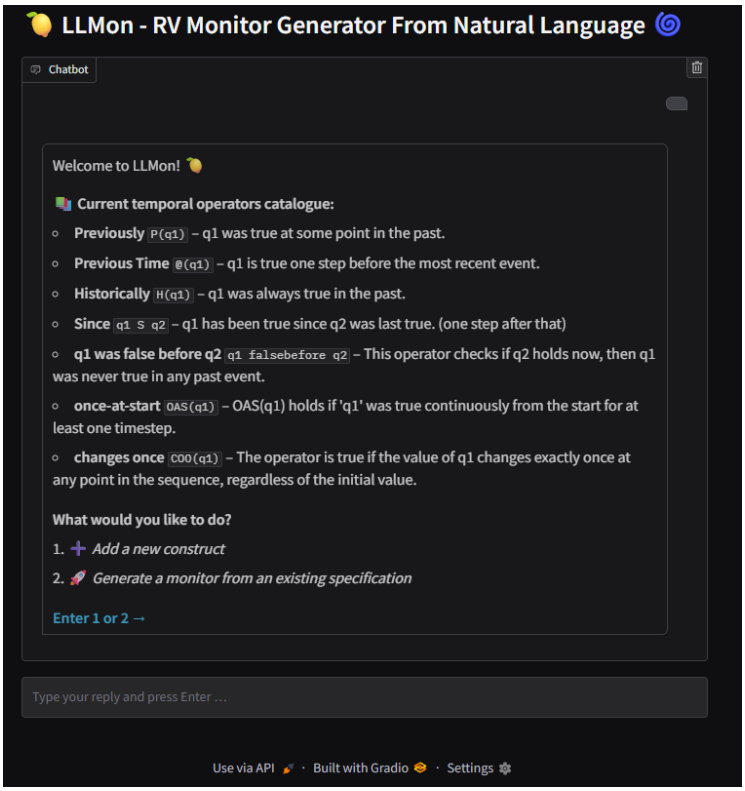

Fig. 3: A screenshot of the web-interface for LLMon.

B Prompts used by LLMon

The full prompts used by LLMon are shown below. Note that some of them contain placeholders for user inputs.

```
Below is the name, notation, and natural language description of a temporal
operator, which is an extension of past-time LTL:
<user's construct input is inserted here>

We aim to calculate the boolean value of this operator, given a sequence of
assignments to the boolean variables, which are the operand/operands of this
operator. Every set of assignments for all operands is considered a single
"event". When a new event arrives, we want to update the truth value of the
entire operator based the content of that event and previous information.
The first received event is considered as the first in the event sequence,
and the most recent event is considered as the last in the sequence.

Create four different python codes that calculate the truth value of the
operator, once a new event arrives. Each code should be based on a slightly
different natural language interpretation of the operator description.
```

```
Important: The code should update the truth value of the operator in an
efficient way, using finite domain variables.

- You may store any additional values that may help to successfully
calculate the truth value. These variables should also have finite domains.

- Number the different codes and clearly explain the interpretation they
represent. The explanation should reliably reflect the related code.
```

Listing 3: Based on the new construct description that is provided by the user, this prompt generates four different natural language interpretations that may slightly differ from the original meaning. In addition, for each new interpretation, the prompt requires generating a Python code that updates the truth value of the construct upon the arrival of a new event. The codes are required to make an efficient update. The different versions are also numbered for future reference. One could also imagine asking the LLM to indicate a preferred interpretation, but we do not adopt this design and instead leave the choice to the user.

```
Generate python code that compares the primary version to all the other
versions, based on all the possible traces of length <k>.

When you mention a version, do not forget to also mention its number. For
each pair, the code should print all the distinguishing traces in the order
of the comparisons (start with all the distinguishing traces of the first
comparison etc.), and for each such trace, print which code version accepts
its (final verdict is true) and which one rejects (final verdict is false).

The first line that the script should print is the version name. When a new
comparison begins, write a caption that corresponds the comparison.
Important: use the following format to display distinguishing sequences:
(replace variable_name, boolean values and formula numbers with the actual
names and values)
"Trace: <variable_name> = (T, T, T); <variable_name> = (T, F, T) => Version
1: True, Version 3: False". If two versions have no distinguishing traces,
clearly mention that the two versions are equivalent. Do not use pandas.

The code should be a fenced Python code block, e.g.:

‘‘‘python
<code here>
‘‘‘
```

Listing 4: The LLM here is instructed to generate a code that compares the primary version selected by the user to the other versions. The pairwise comparison should return all the short traces of length k (a configurable parameter) that distinguish between the two code versions. Subsequently, the tool will extract the code from the LLM response and execute it, presenting the results to the user.

```
Consider the different trace comparisons: (primary code version vs. other
code versions)

<distinguishing traces that were found by the previous prompt>

Analyze these traces, and provide the key differences between the two codes
in each comparison. You can think it through, but eventually provide the
following information for each comparison that has at least one
distinguishing trace:
- key differences as a short description.
- what the key difference means for each code (briefly explain).
- choose one distinguishing trace that supports the key differences and
briefly explain.
```

```
Important: if two versions are equivalent, just mention that the versions
are equivalent using the phrase "equivalent versions", and do not write
anything else.

When you show your analysis, clearly separate between the different
comparisons with hyphens (---).
```

Listing 5: This prompt contains a placeholder for the distinguishing traces that were returned by the code generated in the previous prompt. The LLM is required to analyze these short traces for each pairwise comparison and provide the key difference between the versions in each pair. Note that the key difference has to be based on the distinguishing traces.

```
The user chose version number <user input> as the correct version for the
operator. For the mentioned temporal operator, create a JSON object with
exactly these keys:

\"op_name\": string, # the operator name, as provided by the user.

\"op_notation\": string, # only the operator letter/word (for example "P")

\"op_description\": string,  # according to the chosen operator version.

\"is_unary\": boolean,  # True if unary, False otherwise

\"class_name\": string, # the name of the class (explained below)

\"class_content\": string  # Python class implementing the operator

The class must define:
- An __init__ method that sets initial auxiliary variables, current verdict
and previous verdict.
- An update(arg) or update(arg1, arg2) method that updates the verdict. make
sure to actively update the verdict after every new event.

Preserve the code you generated earlier for this version. This function will
have one argument for unary operator and two for binary operators.

Return ONLY the JSON, no markdown fences.
```

Listing 6: After the user made a decision regarding the version that best represents the construct description, this prompt asks the LLM to output all the relevant information about the construct version. This also includes the code itself in a structure that can be later used by the tool to generate RV monitors. All information must be returned in a JSON format using predefined keys, allowing the tool to efficiently integrate it into its construct knowledge base.

```
Consider the following set of temporal operators:
<JSON array of constructs created by the tool>

Based on these operators and their symbols, translate the following
specification into a formula:
"<natural language specification inserted by the user>"

Guidelines:
- Detect the events in the specification, and turn each event into a boolean
variable (q1, q2 etc)
- Important: think if the specification is ambiguous or not. specifically,
think if there are two or more similar operator that fits the natural
language description. if it is ambiguous, provide all the possible
translations. if it is not ambiguous, provide the single correct translation.
```

```
- The formula (or formulas) should contain only previously mentioned
constructs, the common logical connectives (and, or, not..) and boolean
variables.
- Important: when you write a formula, use the following notation for a
binary operator: 'q1 BINOP q2' (replace BINOP with the relevant symbol), and
the following symbols for the logical connectives: !, &&, ||, ->, <->.
- You are provided with four simple examples:
1. natural language: "The robot was at the store at all times."
temporal formula:  H(q1)
2. natural language: "Whenever the robot is was at the garage, it is not at
home."
temporal formula:  H(q1 -> !(q2))
3. natural language: "In the past, the robot was at home."
temporal formula:  P(q1)
4. natural language: "the robot was previously at q2, but since then it was
always at q1."
temporal formula: q1 S q2


Your final response should contain a JSON file with a few keys keys:
First key - 'var_mapping' - its value is a string that explains the NL event
and their associated boolean variables.
Second key - 'is_ambiguous' - its value should be either True or False,
depending on whether the specification is ambiguous or not.
Third key - 'formulas' - its value should be a list of the possible
translations (one or more formulas with boolean variables).
```

Listing 7: This prompt guides the LLM to extract Boolean events from the natural language specification, assess whether the specification is ambiguous with respect to the available constructs, and output either a single structured translation or a set of alternative candidate formulas (illustrated via few-shot examples over standard past-time LTL constructs).

C Construct knowledge base example

```
[
  {
    "op_name": "weak since",
    "op_notation": "WS",
    "op_description": "q1 has been true since q2 was last true.
       If q2 was never true, then 'a' must be true for the entire sequence.",
    "is_unary": false,
    "class_name": "WeakSinceStrict",
    "class_content": "class WeakSinceStrict:
                        def __init__(self):
                            self.q2_last_true_index = -1
                            self.q1_since_q2 = True

                        def update(self, left, right):
                            q1 = left
                            q2 = right
                            if q2:
                                self.q2_last_true_index = 0
                                self.q1_since_q2 = True
                            elif self.q2_last_true_index != -1:
                                self.q1_since_q2 = self.q1_since_q2 and q1
                            else:
                                self.q1_since_q2 = self.q1_since_q2 and q1

                            return self.q1_since_q2"
  },
  {
    "op_name": "holds-odd",
    "op_notation": "ODD",
    "op_description": "q1 holds at all odd timesteps.",
    "is_unary": true,
    "class_name": "HoldsOddAll",
    "class_content": "class HoldsOddAll:
                        def __init__(self):
                            self.truth_value = True
                            self.current_timestep = 0

                        def update(self, q1):
                            self.current_timestep += 1
                            if self.current_timestep % 2 == 1:
                                if not q1:
                                    self.truth_value = False
                            return self.truth_value"
  }
]
```

Listing 8: An example of a JSON array with two constructs that were generated by the tool. An array of this format serves as the tool's construct knowledge base.