The Power of Reframing: Using LLMs in Synthesizing RV Monitors

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. We present a method and an interactive tool named LLMon that utilizes an LLM for synthesizing RV monitors from temporal or natural language (NL) specifications. Our focus is on allowing the flexibility of defining further temporal constructs on top of the standard ones; it helps to better customize the temporal formalism for expressing requirements originally given in NL. Our method allows the user to define new temporal operators through interaction with the tool. The LLM reframes the given descriptions of new operators, provided by the user in natural language, and synthesizes code for each alternative description. It performs analysis that compares the alternative generated codes, based on searching for traces with distinguishing verdicts. It then generates verbal descriptions that help the user to select the correct semantic interpretation. The tool can be used to synthesize monitors either directly from specifications in the extended temporal logic, or from NL descriptions.

1 Introduction

We study here the synthesis of Runtime Verification (RV) monitors from Natural Language (NL) descriptions with the help of Large Language Models (LLMs) [3, 14, 16, 17]. Such a monitor can follow the execution of a process, based on captured events, and provide a verdict w.r.t. given requirements. The requirements are often written using a fixed formalism such as temporal logic, automata, or rule systems [4–6, 12, 15]. In typical scenarios, users who wish to utilize RV 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*.

In this work, we focus on the translation of requirements that are originally formulated in natural language into RV monitors. This poses some challenges. The natural language description may contain phrases that are far removed from the constructs of the formalism used by a given tool. This may result in unreliable translations. Further, a natural language description may contain some

---

* 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.

ambiguity. Even for simple temporal constructs there can sometimes be alternative interpretations that make sense. Suppose that we want to use a temporal operator $q_1 HB q_2$, where HB stands for *happened before*, which we will use as a running example. One interpretation can be that if q_2 holds then q_1 has happened before or at the same timestep. An alternative interpretation is that q_1 has happened strictly before q_2.

In this work, we present a method and the interactive translation tool LL-Mon [1] based on using LLMs combined with RV synthesis algorithms (e.g., [13]). It can be used to (a) define and implement a monitoring logic in a stepwise 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 [8–10], translating requirements to an existing fixed temporal formalism. A fundamental feature of our method is that it allows the flexibility of *adding new operators* to the temporal logic used as specification formalism. It synthesizes a monitor from a specification that uses these constructs. The semantics for these new operators is acquired through interaction in natural language between our tool and the user, rather than only through formal artifacts. One can consider many 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.

We exploit in our method the fact that the ambiguity that appears in natural language is also inherent in LLMs, which are trained on human texts. This ambiguity is a main challenge in automatically translating natural language descriptions into a formal notation. The fact that the ambiguity weakness is shared by both humans and LLMs is turned by our method 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, including code for each of these. Based on the user's preferences, the tool compares pairs of such interpretations by synthesizing code for the different interpretations, and compares exhaustively the verdicts they provide on short traces. With such *distinguishing traces*, which possess *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.

We limit the target specification logic to propositional constructs with past-time semantics, encompassing past-time propositional LTL, but allowing also user-defined constructs that are not directly expressive in this logic (e.g., a past version of Wolper's example in [18]). Our approach is compositional in that the code generated from individual constructs can be combined with code that is independently constructed for other constructs and form a composite specification. Besides using the newly defined constructs, obtained through interaction with the tool in a structured formula, we can include them as part of an NL requirement, which can be translated to a formula and monitored by the code synthesized by our tool. Ultimately, the use of the suggested methodology and

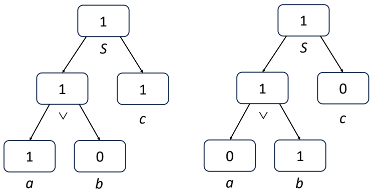

Fig. 1: Syntax tree and summary for $((a \vee b)\,\mathcal{S}\,c)$, given the trace $\langle a = 1,\, b = 0,\, c = 1\rangle; \langle a = 0,\, b = 1,\, c = 0\rangle$

tool 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.

2 Preliminaries

Our method uses LLMs to facilitate synthesizing monitors for temporal specifications. It combines the abilities of LLMs with RV techniques. Specifically, it generates monitors in the style of the past-time LTL monitors in [13]. We describe here that logic and the classical RV algorithm for it.

Propositional Past-Time Linear Temporal Logic. Propositional past-time linear temporal logic (PLTL) is a specification formalism that allows expressing safety properties [2]. 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 and φ has been holding since. The property $\ominus\varphi$ (previous-time φ) means that φ is true in the current trace if φ is true in the trace obtained from the current one by omitting the last event. We can also define the following additional derived operators: $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondminus\varphi = (\mathit{true}\,\mathcal{S}\,\varphi)$ ("past" or "once"), and $\boxminus\varphi = \neg\diamondminus\neg\varphi$ ("always in the past" or "historically").

The Synthesis Algorithm.

The synthesis algorithm for past-time LTL translates the formula into a syntax tree. The monitor uses a *summary* with one bit per each node of the tree, corresponding to a subformula. The truth value of each such bit corresponds to the truth value (0 or 1) of each subformula after the current sequence of observed events.

Upon intercepting a new event, the summary is updated. The updates propagate bottom up according to the syntax tree and depending on the Boolean values of the new event. Figure 1 depicts the syntax tree and summary generated for the formula $\varphi = ((a \vee b)\,\mathcal{S}\,c)$. The left tree is the summary after observing an event with the Boolean values $\langle a = 1,\, b = 0,\, c = 1\rangle$, and the right tree is the summary after further observing an event with $\langle a = 0,\, b = 1,\, c = 0\rangle$. The evaluation of the current right tree will refer to the previous left tree for temporal operator nodes. 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 (left tree), and by $\mathsf{now}(\varphi)$ the value after the update (right tree). The order of updates is bottom up, according to the syntax tree.

```
$\mathsf{now}(\varphi) \leftarrow \mathit{false}$ for each subformula $\varphi$ of $\eta$.
loop
    Observe a new event $s \subseteq A$ as input.
    $\mathsf{pre}(\varphi) \leftarrow \mathsf{now}(\varphi)$ for each subformula $\varphi$.
    for all subformulas $\varphi$ of $\eta$ do
     # If $\varphi$ is a subformula of $\psi$ 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 previously explored alternative way to increase the expressiveness of past-time LTL is by adding *transition rules* to the temporal logic [11]. The rules use new propositions B, disjoint of 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 instance, an example by Wolper [18] shows that one cannot in LTL express that some property holds in every odd state. But this property becomes expressible 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 $\boxminus(q \rightarrow p)$.

3 LLM Synthesis of Monitors with User Defined Temporal Constructs

We present in this section our method for synthesizing RV monitors from NL requirements and the tool LLMon. This involves adding new temporal constructs to a specification formalism. We do this using a running example that showcases the interaction between a prospective user and the tool. We display here only the core content of the prompts given by the tool to the LLM. Note that the user is never interacting directly with the LLM.

Formalism Translation Stage. Assume that the user intends to define a new construct that captures “the occurrence of one event *before* another”. The tool asks the user to provide a name, a notation, and a natural language description for the construct. The input from the user is obtained by the interaction with the tool’s user interface. 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, that asks the LLM to create multiple (by default four) NL interpretations (“reframings”) that are slightly different from the original description. In addition, a Python code is requested in the prompt for every NL interpretation.

```
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.
```

The emphasis on efficiency in the prompt leads the LLM to generate code that updates the construct truth value with minimal memory overhead, aligning 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 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.

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*, which is 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. 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. Here, we assume that the user selected the first interpretation as the primary version. Next, in a separate prompt, 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 (a configurable parameter set up by the user, which should be at least 3) that may differentiate between pairs of the generated code implementations. Since we test the construct codes *in isolation*, it is often sufficient to generate traces of limited length, say with 3-5 events, to capture all the differences in behaviors.

```
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 displays the results (distinguishing traces) to the user. This information is available for the user, who, however, may just pass directly to the next step, where the LLM generates insights on these distinguishing traces.

Tool

Comparison: interpretation #1 vs. interpretation #2
Trace: [(q1=F, q2=F); (q1=F, q2=F); (q1=T, q2=T)] ⇒ V1: T, V2: F
► 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. 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. This explanation is more intuitive for the user than inspecting the distinguishing traces directly, enabling them to understand how their primary interpretation compares to the alternatives. In the listing below we show an example of such a generated text for a single pair of interpretations.

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. Now, the tool prompts the LLM to return relevant details about the selected construct version in a JSON format, structured for seamless integration into the tool's knowledge base. The stored attributes 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 an unary operator and two for a
binary operator.
```

Once the LLM returns all 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. At this point, the user may choose to define another construct or continue to the next stage.

NL Specification Translation Stage. This stage focuses on translating a user-provided natural language specification into a structured formula built from the previously introduced constructs. To offer greater flexibility, the user can choose to describe the specification either in natural language or as a structured formula. If the specification is already provided as a structured formula, this stage is bypassed, and the tool moves directly to the monitor synthesis stage.

To handle the translation, the tool issues a dedicated prompt to the LLM that contains the notations and descriptions of the new supported constructs. Given that the new constructs are defined in natural language, the LLM is able to align requirements with those definitions and produce structured formulas using the constructs. This prompt also contains three few-shot examples of illustrative input-output pairs, as done in [7]. These examples focus on the four standard past-time LTL constructs and primarily illustrate the expected output format given a natural language specification. We demonstrate how the tool applies this dedicated translation prompt using the natural language specification shown below.

User

"Whenever a user writes to a file, it must have been created beforehand. The same requirement applies to reading from a file."

The tool appends this specification, together with the JSON array that was built for the constructs, to the translation prompt. The prompt contains two sequential tasks for the LLM to carry out: (1) identify events in the natural language specification and associate them with Boolean variables and (2) translate the NL specification into a structured formula over the available constructs and the Boolean variables that represent the different identified events. Below is the LLM translation 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 reads from a file.
The specification formula: $(q_1$ HB $q_2)$ && $(q_1$ HB $q_3)$

Monitor Synthesis Stage. The tool generates an RV monitor from the structured formula, using a synthesizer component that integrates the features of the new constructs into the classical algorithm. The synthesizer component utilizes a pre-defined parser that extracts all construct symbols and their arities (assumed to be unary or binary) from the tool's knowledge base. This allows it to correctly parse formulas that include newly added constructs and generate an abstract syntax tree (AST). For each construct encountered, the tool retrieves and embeds its corresponding Python code (describing its update rule) into the synthesized monitor file. The monitor generation then proceeds largely in line with the classical algorithm.

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

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

4 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 [11], 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 a temperature value of 0, we found that introducing slight randomness encouraged more useful interpretation suggestions compared to fully deterministic outputs. 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 in Table 1, we aimed at verifying their correctness: 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. The formulas varied in size, ranging from ten to twenty syntax tree nodes. We first used the tool's synthesizer component to generate RV monitors for each formula. Then, for each generated formula, we created an equivalent formula by replacing the new construct with its corresponding past-time LTL expression. We utilized the tool's synthesizer component again to generate an additional RV

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 used in our experiments. ($\ominus true$ means that there *is* a previous state)

monitor, given the past-time LTL formula where our constructs were translated into their standard past-time LTL form. We compared the two 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, to properly verify them, we had to be able to synthesize RV monitors that support *transition rules*, as presented in Section 2. We extended the tool's synthesizer component to support auxiliary Boolean variables, each defined by a transition rule. After verifying the correctness of each construct as done with the previously mentioned constructs, no discrepancies were observed.

5 Discussion

This work explored a stepwise LLM-assisted workflow for building RV monitors directly from natural language requirements. LLM translation is not perfect. Thus, our approach keeps the human in the loop for selecting the semantics of suggested temporal constructs. It exploits a procedure that facilitates the selection between alternative semantic interpretations for the new constructs. This is based on comparing the verdicts provided on input traces by the different alternative interpretations. Although our method is designed to reduce the reliance on the code generation capabilities of the LLM by performing smaller, human-assisted steps, such a use of LLMs is still not entirely resilient. In our experiments, however, the LLM code generation performed quite reliably. We did not evidence any issues with the LLM automatic generation of code used for the tool to make its internal comparison analysis[3]. Additionally, we observed no issues with the tool's ability to translate NL requirements using the constructs from our experiments. However, we acknowledge that performance in this step may decline if more constructs are introduced or if less capable LLMs are used. Also, the translation of NL requirements to temporal logic may potentially be flawed, which however, may not be much different from how LLMs are used today in code generation.

---

[3] We could have guaranteed further reliability by programming these parts, which are not dependent on the user input, ourselves, although we decided to benefit from the LLM's ability to synthesize this code.

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. 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)
5. 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)
6. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
7. Cohen, I., Peled, D.: End-to-End AI generated runtime verification from natural language specification. In: Steffen, B. (ed.) Bridging the Gap Between AI and Reality. pp. 362–384. Springer Nature Switzerland, Cham (2025)
8. 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)
9. 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)
10. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. CoRR **abs/2206.01962** (2022)
11. 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)
12. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods Syst. Des. **56**(1), 1–21 (2020)
13. 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)
14. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., et al.: Mistral 7B. CoRR **abs/2310.06825** (2023)
15. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Methods Program. **78**(5), 293–303 (2009)
16. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. OpenAI (2018)
17. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., et al.: LLaMA: Open and efficient foundation language models. CoRR **abs/2302.13971** (2023)
18. 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)