Automating Requirements Formalization: Using LLMs and Low-Complexity Distinguishing Traces for Semantic Validation

Daniel Mendoza
Stanford University
dmendo@stanford.edu

Anastasia Mavridou
KBR Inc. / NASA Ames
anastasia.mavridou@nasa.gov

Andreas Katis
KBR Inc. / NASA Ames
andreas.katis@nasa.gov

Caroline Trippel
Stanford University
trippel@stanford.edu

Abstract

Translating natural language (NL) requirements into formal specifications is critical for verifying safety-critical systems, but it is error-prone and time-consuming when done manually. While Large Language Models (LLMs) can automate this translation, they often produce incorrect outputs that require extensive validation. In this paper, we propose ARTEMIS, an LLM-based framework that translates unstructured NL requirements into formal temporal logic (TL) specifications. Our framework reduces validation effort through three synergistic, automated techniques: (i) LLM Translation to Structured NL: We use LLMs to translate unstructured NL requirements into structured NL, which has an unambiguous mapping to TL. This intermediate representation reduces translation errors. (ii) Sub-Specification Generation: We generate low-complexity execution traces (i.e., system behaviors) that correspond to candidate specification fragments from the LLM translations. Users inspect these and accept or reject them. (iii) Balanced Distinguishing Trace Generation: We minimize the number of traces users need to inspect by pruning the candidate specification space. Each accepted or rejected trace eliminates candidates logarithmically. We evaluate ARTEMIS on five real-world safety-critical requirements datasets. The results show that it achieves 1.57X higher translation accuracy while reducing manual validation effort by up to 10.83X compared to state-of-the-art baselines.

CCS Concepts

• **Software and its engineering → Formal methods**; **Requirements analysis**; • **Computing methodologies → Natural language processing**.

Keywords

requirements formalization, temporal logic, large language models

ACM Reference Format:
Daniel Mendoza, Anastasia Mavridou, Andreas Katis, and Caroline Trippel. 2026. Automating Requirements Formalization: Using LLMs and Low-Complexity Distinguishing Traces for Semantic Validation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787815

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787815

1 Introduction

Thorough verification is required to guarantee, or enhance confidence, that software systems will behave correctly. It is especially important for mission- or safety-critical systems (e.g., healthcare [16], aeronautics [14], aerospace [11], automotive [51], robotics [50, 54]), where failures can lead to the disruption of critical missions or pose serious risks to human safety.

Whether performed dynamically at runtime or statically using formal methods (e.g., model checking [7]), thorough verification relies on formal *specifications* to precisely define a system's allowed behaviors. Such specifications are typically expressed as TL formulas (e.g., using linear temporal logic (LTL) [47]), where each formula encodes a set of allowed execution traces (i.e., system behaviors). System *requirements*, however, are usually first written as plain English (unstructured NL) sentences [14, 16, 17, 23, 25, 28, 44, 49, 50, 57]. Unfortunately, **manual translation** of requirements to specifications is cumbersome and error-prone, even for formal methods experts, limiting the use of thorough verification in practice [49].

Recently, automated generation of specifications from unstructured NL requirements has emerged as a compelling application of natural language processing (NLP) methods, especially large language models (LLMs) [6, 8, 10, 15, 19, 20, 27, 28, 33–35, 37, 38, 41, 52, 55]. These **generate-and-validate** approaches generate candidate specifications for a given requirement and, as in manual translation, rely on manual validation to disambiguate them. The need for manual validation stems partially from the inherent ambiguity of unstructured NL, which often leads to multiple plausible candidate specifications per unstructured NL requirement. Plus, NLP methods can output blatantly erroneous specifications. To (hopefully) cover many plausible specifications, generate-and-validate typically produces many more candidates than manual translation; thus, which approach demands the least user effort remains unclear.

To manually validate candidate specifications, users inspect them [4, 10, 41] or execution traces they admit [20, 24], possibly with NL explanations [24]. Though, analyzing traces is more intuitive; inspecting TL formulas can be as hard as writing them. Existing trace-based approaches produce either *simulation traces* for a single candidate [24] or *distinguishing traces* [20] between *pairs* of candidates for a user to accept/reject. All produce traces that cover *full* candidate specifications, often involving many variables, making them complex and difficult to assess. While distinguishing traces directly disambiguate candidates, existing methods may require inspecting $O(n)$ traces to converge on the correct candidate among n candidates. This becomes impractical for large n, e.g., for generate-and-validate which often produces many candidates.

This Paper. We propose **ARTEMIS** (Automated Requirements Translation and Evaluation with Minimal Interactive Supervision), an LLM-based generate-and-validate framework that significantly

reduces manual effort compared to manual translation and existing generate-and-validate approaches. ARTEMIS comprises three synergistic, automated techniques.

First, with *LLM Translation to Structured NL*, ARTEMIS uses LLMs to generate *structured* NL requirements, rather than specifications directly, from unstructured NL requirements. Structured NLs [2, 24, 26, 32], originally designed to support manual translation of requirements to specifications for non-experts, employ restricted grammars with unambiguous mappings to TL. Using structured NL as an intermediate representation (IR) reduces erroneous candidate specifications by leveraging LLMs' stronger reasoning in NL compared to TL. Consequently, LLMs generate fewer candidates to produce plausible ones, reducing manual validation effort (§7.2).

Next, ARTEMIS introduces *Sub-Specification Generation* to produce distinguishing traces for *fragments* of full candidate specifications derived from structured NL requirements. Sub-Specification Generation is enabled by decomposing a structured NL requirement into a tree, where child requirements are sub-requirements of parent requirements. Because structured NLs define a finite number of specification templates (§2.2), ARTEMIS can derive a *proxy specification* for each sub-requirement to capture some permissible behavior of all full candidate specifications that could ever contain it, restricted to the sub-requirement's variables. Distinguishing traces are then generated for proxies, producing traces with fewer variables compared to generating traces for full specifications (§7.3).

Finally, with *Balanced Distinguishing Trace Generation*, ARTEMIS reduces the worst-case number of traces that a user must inspect to disambiguate n candidate specifications from $O(n)$ in prior work [20] to $O(log(n))$. It achieves this by generating distinguishing traces that eliminate approximately half of all candidates when accepted or rejected by a user.

Overall, this paper makes the following contributions:

- **LLM Translation to Structured NL:** We show that structured NL IR reduces translation errors, and thus manual validation effort, when automating specification generation with LLMs.
- **Sub-Specification Generation:** By using structured NL as a translation IR and decomposing candidate structured NL requirements into sub-requirements, we enable generation of proxy specifications, which produce low-complexity traces.
- **Balanced Distinguishing Trace Generation:** We develop an efficient algorithm that enables a user to select one specification out of n candidates by inspecting $O(\log(n))$ traces.
- **ARTEMIS:** By combining the three techniques above, we establish, for the first time, LLM-based generate-and-validate as the most automated approach to formalizing NL requirements.
- **Evaluation:** We use ARTEMIS to formalize real-world unstructured NL requirements from five datasets, covering safety-critical domains like aeronautics [14], health care [16], and robotics [50]. Compared to state-of-the-art baselines [6, 10, 20, 28, 41], ARTEMIS achieves 1.57X higher translation accuracy and, for requirements that have more than 10 candidate specifications, reduces manual validation effort by 2.52X on average (up to 10.83X).

2 Background and Related Work

We provide background on LTL (§2.1), review prior work translating unstructured NL requirements to TL specifications (§2.2)—including FRETish [24], a structured NL we use to demonstrate ARTEMIS's translation IR—and discuss manual validation methods (§2.3).

2.1 Linear Temporal Logic Specifications

LTL [47] expresses properties over execution *traces*, where a trace is a sequence of system states over time. LTL formulas are built over a set of *atomic propositions* (evaluating to *true* or *false*), Boolean constants ($\top$, $\bot$), as well as logical operators of propositional logic: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and $\rightarrow$ (implication). LTL extends propositional logic with temporal operators: $X\phi$ (next) asserts ϕ holds in the *next* state; $F\phi$ (eventually) asserts ϕ holds in *some* future state; $G\phi$ (globally) asserts ϕ holds in *all* future states; $\phi U \psi$ (until) asserts ψ holds in some future state, and ϕ holds at every future state up to the state where ψ holds. For example, $\neg p \wedge XFp$ holds for trace $\neg p; (\neg p; p)^{\omega}$, where ";" separates timesteps and "$()^{\omega}$" indicates an infinitely repeating sequence.

2.2 From Requirements to Specifications

System requirements are typically written in unstructured NL (§1). We next discuss prior work that translates unstructured NL requirements to TL specifications via (1) **manual translation** with structured NL and (2) **generate-and-validate** with NLP methods (see **Prior Work:Translation** box in Fig. 1).

Manual Translation with Structured NL. Manually translating unstructured NL requirements into formal specifications is an error-prone and time-consuming task, even for experts. To support this process, Specification Pattern System (SPS) [13] and Easy Approach to Requirements Syntax (EARS) [39] observe that specifications across domains often follow recurring patterns. Building on this insight, structured NLs such as FRETish [24], SPIDER [32] and PSP [2] define restricted NL grammars with deterministic mappings to LTL specifications, enabling users to conveniently capture common patterns. A Structured NL grammar defines a finite set of fields, each capturing a different requirement aspect. Each field offers template options that express distinct TL semantics. A structured NL requirement instantiates these fields by selecting a template option for each field and filling in its parameters. W.l.o.g., we use FRETish as an IR throughout the paper to illustrate our approach for ARTEMIS and use both FRETish and PSP for evaluation in §7.

A FRETish requirement contains up to six fields ('*' = required):

(1) `scope` specifies time intervals where the requirement is enforced, e.g., "while scope," where scope is a Boolean expression;
(2) `condition` triggers a `response` subject to `timing` to occur at the time that the Boolean expression keyword first evaluates to true with "upon keyword," or every time keyword evaluates to true with "whenever keyword;"
(3) `component*` is the system component upon which the requirement is levied;
(4) `shall*` is used to express that the component's behavior must conform to the requirement;
(5) `timing` specifies when `response` shall happen, e.g., "immediately," "at next point," "eventually," "within N ticks," subject to the constraints defined in `scope` and `condition`; and
(6) `response*` is the Boolean expression that the component's behavior must satisfy.

In the paper, we omit `component` and `shall` for brevity, since they do not impact the specification semantics. We also refer to `timing` and `response` together as `timing`. For example, a plausible translation of the unstructured NL requirement in Fig. 2 is: "while !standby & support & !fail, upon limit, at next point pullup."

A FRETish requirement may have multiple `condition`s, but at most one of each `scope` and `timing`. Since `timing` is interpreted relative to a `condition`, which is contextualized by a `scope`, FRETish fields always follow the following order: `scope` → `condition` → `timing`. The algorithm underlying the translation of FRETish structured NL into LTL is detailed in prior work [24].

Automatic Translation with NLP. Recent NLP advances have inspired numerous efforts to automatically generate TL specifications *directly* from unstructured NL, e.g., using LLMs [6, 8, 10, 15, 19, 27, 28, 33–35, 37, 38, 41, 55] and related methods [20, 52]. Compared to human translators, NLP-based translators often enumerate many more *candidate* specifications to cover one that matches the user's intent, since they are prone to producing erroneous outputs.

2.3 Manually Validating Specifications

Manual validation is standard for resolving unstructured NL ambiguity and errors when formalizing requirements. It enables a domain expert to converge to one *plausible* specification—one consistent with their interpretation of the NL. A requirement is *ambiguous* if there are multiple semantically different plausible specifications according to different experts. A specification is *erroneous* if there is no expert that would deem it plausible. Unstructured NL is inherently prone to ambiguity, e.g., "*latch an autopilot pullup*," which appears in our running example (§3, Fig. 2), is ambiguous regarding when to assert *pullup* and for how long.

Prior work (see **Prior Work: Validation** box in Fig. 1) proposes manually validating candidate specifications by inspecting them or traces they admit.

Inspecting Specifications. NLP-based automated translators [6, 8, 10, 15, 19, 20, 27, 28, 33–35, 37, 38, 41, 52, 55] ask users to manually validate full specifications. Prior work asks users to inspect full candidates and manually construct a counterexample trace when a candidate does not match their interpretation of the requirement [4]. Upon receiving a counterexample, it leverages *specification learning* [4, 18, 20, 22, 43, 45, 48, 56] to re-generate a candidate specification consistent with new and previous counterexamples.

Recent work [10, 41] proposes to make LLM-generated specifications easier to inspect. nl2spec [10] employs *unstructured* translation decomposition, which instructs LLMs to decompose a translation task into simpler sub-tasks and combine their results into a full specification. Then, it asks the user to inspect the generated *sub-specification* of each sub-task and the full specification.

SynthTL [41] decomposes the same translation task into a *logical combination* of sub-tasks (i.e., *structured* translation decomposition), which produce sub-specifications that mechanically compose into a full specification. The user inspects sub-specifications and their logical structure, but never the full specification. However, SynthTL attempts to overlay the structure of TL on unstructured NL to perform decomposition, which restricts compatible inputs (§7.2). Nevertheless, validating specifications (even sub-specifications [41]) by direct inspection can be as hard as writing them.

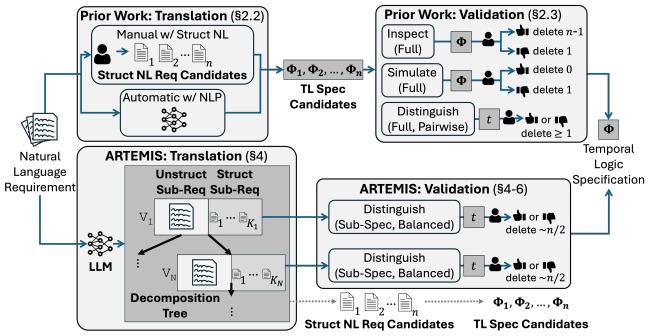


Figure 1: Prior work and ARTEMIS translate an unstructured NL requirement to a TL specification Φ. Prior work produces candidate specifications $\Phi_1, ..., \Phi_n$ manually using structured NL or automatically using NLP methods. The user validates candidates via direct inspection, simulation, or pair-wise distinguishing trace validation. Using LLMs, ARTEMIS automatically decomposes and translates sub-requirements that collectively imply full structured NL candidates. The user validates the candidate specifications they imply by inspecting fewer sub-specification-level distinguishing traces.

Inspecting Traces. Manually determining whether *one* trace represents an allowed system behavior is often easier than determining whether a specification, which encodes a set of traces, precisely captures *all* allowed behaviors.

FRET [24] uses *simulation* to produce traces that belong to one specification, but, this does not directly disambiguate candidates.

LTLTalk [20] produces *distinguishing traces*, each of which distinguishes at least two candidate TL specifications. To generate a distinguishing trace, LTLTalk collects the most likely pair of candidates (Φ_1, Φ_2) (according to an NLP model), and uses a model checker to find a satisfying trace for the formula $(\Phi_1 \wedge \neg\Phi_2) \vee (\neg\Phi_1 \wedge \Phi_2)$. By accepting/rejecting such a trace, the user is guaranteed to discard at least one candidate. Given n candidate specifications, LTLTalk can supply up to $n-1$ traces to the user to converge to a single correct specification, which is likely impractical for large n.

In general, manually determining whether a trace is allowed can be difficult for complex requirements involving many variables.

Challenges for Generate-and-Validate. The burden of manual validation renders it unclear whether generate-and-validate offers any advantage over manual translation. To see why, suppose a translation approach produces $n = n_p + n_e$ candidate specifications, where n_p (n_e) denotes the number of plausible (erroneous) candidates. To converge to a single correct specification, the user must inspect up to $O(n_p + n_e)$ TL formulas or traces. That is, manual effort increases linearly in the worst case for each erroneous candidate. In contrast, expert human translators are unlikely to make blatant translation errors, and typically only consider plausible specifications. So, manual translation requires $\sim O(n_p)$ effort.

3 ARTEMIS: Approach and Prior Work

We now present ARTEMIS, our generate-and-validate framework for translating unstructured NL requirements to LTL specifications.

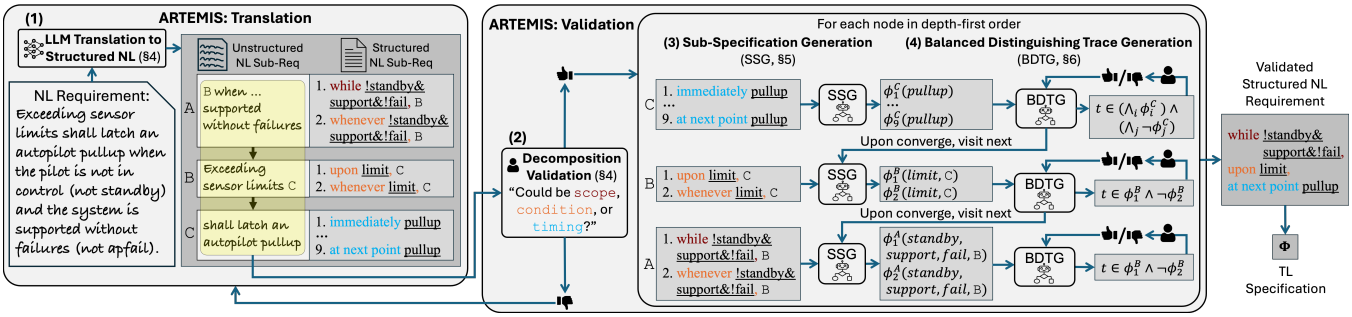


Figure 2: NL to TL Translation with ARTEMIS: (1) generate a decomposition tree mapping unstructured sub-requirements to candidate structured sub-requirements; (2) query the user to validate the decomposition of unstructured sub-requirements. If rejected, return to (1). Otherwise, in depth-first order, for each node, (3) derive a proxy for each candidate, (4) generate distinguishing traces, query the user to accept/reject the traces, and prune candidates inconsistent with user responses. When no distinguishing traces remain for a node, proceed to the next; after visiting all nodes, the remaining candidates compose a single full TL specification.

We organize the discussion into two main parts corresponding to the key phases of the approach:

- **Translation** (§3.1): How ARTEMIS generates candidate specifications using LLMs and structured NL as an intermediate representation compared with prior work (§2.2).
- **Validation** (§3.2–§3.3): How ARTEMIS enables users to validate candidates through low-complexity, low-effort distinguishing traces, compared with prior work (§2.3).

Fig. 1 provides a high-level comparison of ARTEMIS (bottom) with prior work (top). While prior work either requires manual translation to structured NL or uses NLP methods to generate TL directly—both followed by validation of full specifications or full-specification traces—ARTEMIS automatically generates structured NL and validates using sub-specification-level traces. This combination enables ARTEMIS to achieve higher translation accuracy while requiring fewer, simpler traces for validation. For $n = n_p + n_e$ candidates with n_p plausible and n_e erroneous, prior generate-and-validate requires $O(n)$ manual effort and manual translation requires $O(n_p)$ (§2.3), whereas ARTEMIS requires $O(\log(n))$ manual effort, establishing generate-and-validate as the most automated approach (when n_e is subexponential in n_p).

Fig. 2 illustrates ARTEMIS's complete workflow on a real-world requirement from the LMCPS dataset [14], which serves as our running example throughout the paper.

3.1 LLM Translation to Structured NL

ARTEMIS introduces *LLM Translation to Structured NL* (§4), which uses LLMs to produce candidate structured NL requirements that deterministically map to TL specifications. Concretely, this step takes as input an unstructured NL requirement and uses LLMs to automatically generate a *decomposition tree*, where each node maps unstructured NL sub-requirements to candidate structured NL sub-requirements. For example, on the left side of Fig. 2 (see step 1: **LLM Translation to Structured NL**), the input requirement "*Exceeding sensor limits [..] failures (not apfail).*" is decomposed into three nodes (A, B, C), each representing an unstructured sub-requirement. Node B contains the unstructured sub-requirement "Exceeding sensor limits", which the LLM translates into two candidate structured sub-requirements "upon limit" and "whenever limit" both instances of the FRETish `condition` field with different semantics.

In step 2 (**Decomposition Validation**, Fig. 2), ARTEMIS asks the user to confirm that all *unstructured* sub-requirements (one per node): (i) do not lose/hallucinate information from the input requirement and (ii) could conceivably map to *some* structured NL field (e.g., FRETish `scope`, `condition`, or `timing`). If an unstructured sub-requirement is rejected for a node, the user can either ask ARTEMIS to regenerate the entire tree or manually fix it and have ARTEMIS regenerate candidates only for that node.

Unlike prior LLM-based approaches that translate directly to TL [6, 8, 10, 15, 19, 27, 28, 33–35, 37, 38, 41, 55], ARTEMIS's translation to structured NL reduces erroneous candidate specifications and enables decomposed translation without input restrictions.

3.2 Sub-Specification Generation

After Decomposition Validation passes, ARTEMIS visits decomposition tree nodes in depth-first order and, to enable generating traces that distinguish the node's candidate structured sub-requirements, derives a TL specification for each candidate (step 3: **Sub-Specification Generation**, Fig. 2). Existing structured NLs [2, 24, 26, 32] only define TL semantics for full requirements, i.e., they do not provide semantics for individual fields or partial instantiations in isolation. Yet, generating distinguishing traces for sub-requirements requires defining their TL semantics independently.

ARTEMIS's Sub-Specification Generation automatically derives a TL specification, called a *proxy*, that captures each candidate's local TL semantics (detailed in §5). A proxy for a candidate structured sub-requirement (i) *omits ancestor variables*: excludes variables from parent nodes in the decomposition tree and (ii) *optionally omits descendant variables*: represents each child node as an atomic proposition instead of including variables from the child and its descendants. In Fig. 2, consider node B with candidates "upon limit, C" and "whenever limit, C". Proxies $\phi_1^B(limit, \mathtt{C})$, $\phi_2^B(limit, \mathtt{C})$ for these candidates (i) omit variables from B's parent node A (i.e., *standby*, *support*, *fail*), (ii) omit variables from B's child node C (i.e., *pullup*),

and (iii) use an atomic proposition C to abstractly represent node C's semantics. As a result, traces for node B involve only *limit* and C, making them simpler to inspect. Similarly, for node C with candidates "immediately pullup" and "eventually pullup," proxies $\phi_1^C(pullup), ..., \phi_9^C(pullup)$ involve only one variable *pullup*.

Comparable prior work generates traces over full specifications involving all variables [20]. In contrast, proxies enable generating traces localized to specific sub-requirements, producing simpler traces with fewer variables and localizing manual inspection effort to one sub-requirement at a time.

3.3 Balanced Distinguishing Trace Generation

After deriving proxies for a node, ARTEMIS runs **Balanced Distinguishing Trace Generation** (BDTG) as shown in step 4 of Fig. 2. BDTG (i) generates distinguishing traces between the node's candidate proxies; (ii) asks the user to accept/reject each trace based on their understanding of the node's unstructured sub-requirement; and (iii) prunes candidates whose proxies contain (do not contain) a rejected (an accepted) trace. For example, for node C in Fig. 2, the user is queried to accept/reject traces according to their understanding of unstructured sub-requirement "shall latch an autopilot pullup." Thanks to proxies, all traces for node C involve only the variable *pullup*, and thus, are easier to validate.

Once no distinguishing traces can be generated for a node, ARTEMIS continues to the next node. This happens either when there is only one remaining candidate for the node or all of its candidates have the same proxy. From visiting all nodes, ARTEMIS converges to a full Validated Structured NL Requirement (see right-hand side of Fig. 2), and produces the corresponding TL specification as defined by the structured NL.

Prior work produces distinguishing traces between *pairs* of candidate specifications [20], requiring $O(n)$ traces in the worst case. ARTEMIS's BDTG considers *all* candidates simultaneously and aims to rule out half with each accepted/rejected trace. This reduces the worst-case number of distinguishing traces needed to converge to a plausible specification from $O(n)$ to $O(log(n))$ (detailed in §6).

Manual Effort. Human validation is a mandatory step in requirements engineering, particularly within safety-critical domains. In these areas, such as avionics (e.g., regulated by standards like DO-178C [11]), even small specification errors can lead to catastrophic failures [3]. This necessity imposes a significant manual validation burden. ARTEMIS is designed to reduce this manual burden while maintaining the necessary human-in-the-loop oversight mandated by industry regulations. Manual effort in ARTEMIS is limited to (i) Decomposition Validation: confirming that unstructured sub-requirements faithfully represent the input and (ii) Trace Validation: accepting/rejecting distinguishing traces. However, effort of the former only requires intuitive reasoning over NL (no precise reasoning), which is negligible compared to the latter.

4 LLM Translation to Structured NL

This section details how ARTEMIS's LLM Translation to Structured NL (step 1 in Fig. 2) produces decomposition trees and validates them with users. Recall from §3.1 that this step leverages LLMs' superior reasoning in NL over TL to reduce erroneous outputs.

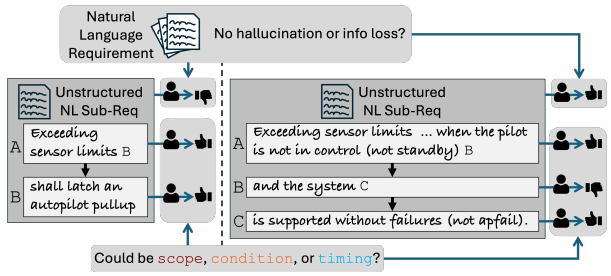


Figure 3: Two decomposition trees for the same requirement rejected during ARTEMIS's Decomposition Validation (§4).

Decomposition Tree Construction. ARTEMIS uses LLMs to construct a decomposition tree where: (i) **nodes** contain an unstructured sub-requirement and its *candidate* structured sub-requirements; (ii) **edges** denote breaking a parent sub-requirement into simpler child sub-requirements and (iii) **child references** are represented by symbols that parents can reference. E.g., in step 1 of Fig. 2, parent node A references its child node B using symbol B. The structured sub-requirement at node A is "while !standby & support & !fail, B," where B is a placeholder for sub-requirement node B.

An unstructured sub-requirement should be a substring of the input unstructured NL requirement, or some NL phrase that is synonymous with such a substring. Only structured sub-requirements (not unstructured ones) must satisfy syntactic restrictions of the structured NL. ARTEMIS produces a decomposition tree in a single LLM prompt, following prompting strategies of prior work that decompose requirements and translate directly to TL [10, 38, 41].

Deriving Full Candidate Specifications. A decomposition tree compactly represents multiple full candidate specifications. We derive them as follows:

(1) Initialize a set with the root node's structured sub-requirements.
(2) For each structured requirement in the set, replace all reference symbols with a structured sub-requirement from its child node.
(3) Repeat until no requirement contains reference symbols.
(4) Remove any ill-formed structured NL requirements.

For example, see step 1 in Fig. 2. Starting at root node A with "while !standby & support & !fail, B," B is expanded to form:

- "while !standby & support & !fail, upon limit, C"
- "while !standby & support & !fail, whenever limit, C"

Then, expanding C yields full candidate structured NL requirements.

Decomposition Validation. To support subsequent trace analysis (step 4 in Fig. 2), users must be able to determine whether a trace is allowed or disallowed with respect to the unstructured sub-requirements. This is possible if the sub-requirements faithfully represent the input requirement without losing or hallucinating information. To this end, users must confirm that (i) the unstructured sub-requirements collectively preserve all information from the input and (ii) each unstructured sub-requirement could plausibly map to one or more structured NL fields.

Fig. 3 zooms into step 2 of Fig. 2 showing two rejected decomposition attempts (left and right) for our running example.

- Left example (information loss): Each unstructured sub-requirement (of nodes A and B) is confirmed to map to a FRETish structured NL field (thumbs up on each node). However, the user rejects

the overall decomposition (thumbs down, top left) because information is missing: no sub-requirement captures the excerpt "*when the pilot is not in control [..] without failures (not apfail)*."

- Right example (invalid field mapping): The decomposition preserves all information from the input requirement without loss or hallucination (thumbs up, top right). However, the user rejects node B's unstructured sub-requirement "and the system C" because it cannot be mapped to any FRETish field type.

In contrast, notice that the decomposition tree illustrated in Fig. 2 passes Decomposition Validation.

Importantly, Decomposition Validation requires low effort and expertise: users need only basic familiarity with structured NL fields (designed to be intuitive) and do not need a detailed understanding of their semantics. Because each unstructured sub-requirement is usually a substring of the input requirement, validating it requires no more effort than straightforward reading comprehension. In our evaluation (§7.3), decomposition trees contained fewer than four unstructured sub-requirements per requirement.

Decomposition Expressiveness. By requiring that structured sub-requirements in a decomposition tree be valid structured NL fields, ARTEMIS effectively uses LLMs to overlay the structured NL grammar on top of an input unstructured requirement. Unlike prior work [41], which overlays TL (§2.3), ARTEMIS's approach retains nearly all expressiveness of the input requirement, eliciting better translation accuracy in practice (§7.2). Intuitively, this is because structured NL is designed to capture common requirement/specification patterns (§2.2). Notably, all requirements in our evaluation are amenable to decomposition by the structured NL fields. Encountering a requirement that cannot be decomposed is likely a sign that it corresponds to a new TL pattern not yet captured by the structured NL, which is likely rare (§7.4). For input requirements that cannot be decomposed, ARTEMIS would produce a tree where each of the nodes' structured sub-requirements correspond to one field of the structured NL grammar, and all unstructured sub-requirements contain the full input requirement.

5 Sub-Specification Generation

Recall from §3.2 that structured NL grammars do not provide TL for structured sub-requirements, which are required to generate traces from them. The TL semantics of a single field (§2.2) often depends on the instantiations of other fields, and thus, cannot be extracted as subformulas of a full TL specification. Hence, isolating field-level semantics is non-trivial and requires careful abstraction. This section details how ARTEMIS solves this problem using the notion of a *proxy*, i.e., a TL specification that captures a structured sub-requirement's local semantics and Sub-Specification Generation (SSG), an automated procedure for deriving them.

Given a decomposition tree with N nodes, each with K structured sub-requirements, prior work [20] must generate distinguishing traces for all $O(K^N)$ full candidate specifications. ARTEMIS only needs to disambiguate $O(K)$ sub-requirements per node.

5.1 Proxy for Semantics of Sub-Requirements

ARTEMIS supports two proxy types: (i) *Standard Proxy*, which omits variables used only in the node's ancestors, and (ii) *Abstract Proxy*,

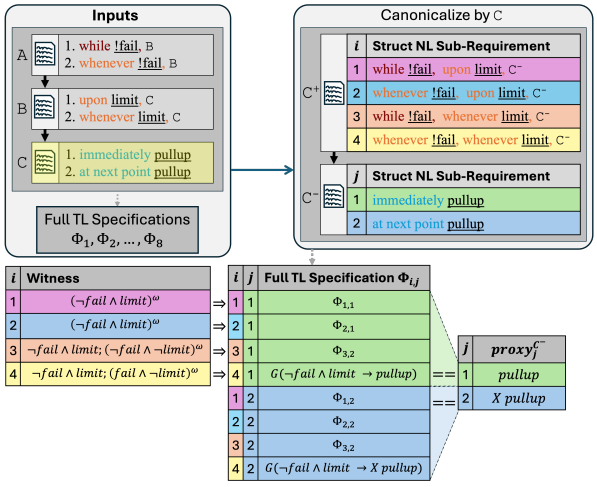


Figure 4: Example of canonicalization and proxies for node C of Fig. 2. Colors denote index i and j correspondence. Standard proxy $proxy_j^{C^-}$ for the j-th sub-requirement of node C^- uses only that sub-requirement's variables and captures behavior common to all specifications $\Phi_{i,j}$ containing it (Eq. 1).

which is a standard proxy that also omits descendants' variables by representing each child node with one atomic proposition.

Fig. 4 shows standard proxies for node C of our running example (Fig. 2). For clarity, the tree in Fig. 4 has been simplified from Fig. 2 by removing variables *support* and *standby* in node A, and shows only two candidate structured sub-requirements for node C.

Canonicalization. To define proxies formally, we first canonicalize the decomposition tree with respect to a target node v. Canonicalization reshapes the tree structure without changing the full specifications it represents:

- All structured sub-requirements of node v and its descendants collapse into a single child node v^-.
- All structured sub-requirements of v's ancestors and their relatives (excluding node v and its descendants) are collapsed into a single parent node v^+.

Let $\Phi_{i,j}$ denote the full TL specification corresponding to pairing the ith structured sub-requirement of parent v^+ with the jth sub-requirement of child v^-.

For example, in Fig. 4, the input tree (top left) is canonicalized with respect to node C into a tree (top right) with parent node C^+ and child node C^-. Both trees represent the same set of full TL specifications, and $\Phi_{1,2}$ corresponds to the full structured NL requirement "while !fail, upon limit, at next point pullup."

Standard Proxy. A standard proxy $proxy_j^{v^-}$ for sub-requirement j of child node v^- contains only variables in sub-requirement j of v^-, and satisfies the following:

$$\forall i.\ \exists t_i^{witness}.\ \forall j.\ \left(t_i^{witness} \Rightarrow \left(proxy_j^{v^-} \leftrightarrow \Phi_{i,j}\right)\right) \qquad (1)$$

That is, for all sub-requirements i in parent node v^+, there exists some witness trace $t_i^{witness}$ over just the variables in v^+, where $proxy_j^{v^-}$ is equivalent to full specification $\Phi_{i,j}$. In other words, $proxy_j^{v^-}$ captures behavior over variables in the j-th sub-requirement

of node v^- that is common to all full specifications $\Phi_{i,j}$ that contain sub-requirement j. Consequently, if a user rejects a trace from $proxy_j^{v^-}$, then for all i in v^+, there exists a trace in $\Phi_{i,j}$ that would be rejected by the user. Similarly, if a user accepts a trace from $\neg(proxy_j^{v^-})$, there exists a trace in $\neg(\Phi_{i,j})$ that would be accepted. Thus, any user response that is inconsistent with $proxy_j^{v^-}$ implies that, for all i, the corresponding full specifications $\Phi_{i,j}$ are also inconsistent and can be safely eliminated.

E.g., for node $\mathbb{C}^-$ in Fig. 4, LTL formulas $proxy_1^{C^-} = pullup$ and $proxy_2^{C^-} = Xpullup$ are proxies for "immediately pullup" and "at next point pullup," respectively. Observe that given valuations of witness trace $\neg fail \wedge limit; (\neg fail \wedge \neg limit)^\omega$ for $i = 4$, all traces that satisfy (dissatisfy) full specification $\Phi_{4,2} = G(\neg fail \wedge limit \rightarrow Xpullup)$ for $i = 4, j = 2$ likewise satisfy (dissatisfy) $proxy_2^{C^-} = Xpullup$ for $j = 2$. This shows that $proxy_2^{C^-}$ captures the behavior over variable $pullup$ exhibited by full specification $\Phi_{4,2}$. This holds for all i of node $\mathbb{C}^+$ and for all j of node $\mathbb{C}^-$, satisfying Eq. 1.

Abstract Proxy. An abstract proxy extends a standard proxy by introducing, for each child of node v in the original tree (before canonicalization), an atomic proposition representing that child and its descendants. This allows variables in descendants to be omitted during trace inspection. Specifically, an abstract proxy $abstract_j^{v^-}$ for node v^- includes the constraint:

$$\bigwedge_{u \in Ch(v)} \bigwedge_{j'} G(proxy_{j'}^{u^-} \leftrightarrow p_u) \tag{2}$$

where $Ch(v)$ is the set of children of node v, p_u is an atomic proposition representing child u and its descendants, and j' ranges over the standard proxies of child u. This ensures that, for every trace satisfying $abstract_j^{v^-}$, at each time step, atomic proposition p_u holds (does not hold) iff all of child u's standard proxies $proxy_{j'}^{u^-}$ hold (do not hold) at the same time step. In other words, given some trace in $abstract_j^{v^-}$, atomic proposition p_u is consistent with the evaluation of all of child u's standard proxies at each time step. Hence, p_u captures the semantics of child u, allowing variables in child u and its descendants to be omitted from traces shown to the user.

5.2 Deriving Proxies Automatically

Given a canonicalized node v^-, SSG produces a standard proxy for each of its structured sub-requirements, from which abstract proxies are optionally derived. SSG uses two sub-routines: Proxy Guess (guesses likely standard proxies) and Proxy Check (confirms they satisfy Eq. 1). In this section, we provide an overview of Proxy Guess and Proxy Check; full details can be found in our artifact [42].

Proxy Guess. For a canonicalized node v^-, ARTEMIS's Proxy Guess inputs the set of full specifications $\Phi_{i,j}$, and guesses a set of standard proxies $proxy_j^{v^-}$ for each of v^-'s candidate structured sub-requirement that likely satisfy Eq. 1. Each guess is a conjunction of a subset of all full specifications $\Phi_{i,j}$. If a standard proxy for a sub-requirement satisfying Eq. 1 exists, Proxy Guess is guaranteed to find it after finitely many guesses. Since Proxy Guess makes only finitely many guesses, SSG always terminates. A standard proxy does not exist only when all full specifications containing the structured sub-requirement exhibit no common traces over its variables (we did not encounter this in our evaluation, §7).

Abstract proxies for a node are derived directly from the node's standard proxies. The abstraction constraint Eq. 2 is unsatisfiable when the standard proxies of the child nodes do not exhibit any common behavior. When the abstraction constraint is satisfiable, ARTEMIS uses abstract proxies to generate distinguishing traces since they often produce traces with fewer variables compared to standard proxies; otherwise, ARTEMIS uses standard proxies.

Proxy Check. For a canonicalized node v^-, ARTEMIS's Proxy Check procedure verifies that a set of standard proxies for v^- satisfies Eq. 1. Proxy Check takes as input: (i) a set of LTL formulas encoding one proxy $proxy_j^{v^-}$ for each candidate j of v^-, (ii) a set of variables Var that are used in v^-'s candidates, and (iii) a set of LTL formulas encoding all full specifications $\Phi_{i,j}$. For each i, Proxy Check outputs a witness trace $t_i^{witness}$ over the variables not in Var, for which all proxies $proxy_j^{v^-}$ and the full TL specification $\Phi_{i,j}$ are equivalent (per Eq. 1). If no witness traces exist, Proxy Check returns $\emptyset$. Note that a standard equivalence encoding between $proxy_j^{v^-}$ and $\Phi_{i,j}$ in a satisfiability model checker query is insufficient to derive $t_i^{witness}$ of Eq. 1, as it yields traces over all variables of all $\Phi_{i,j}$. To generate $t_i^{witness}$ independent from valuations of variables in Var, ARTEMIS introduces the Proxy Check procedure. In contrast, the abstraction constraint for abstract proxies, Eq. 2, can be checked via a standard LTL model checker query.

Offline Proxy Generation. Instead of regenerating proxies for every decomposition tree, SSG generates standard proxies for each sub-requirement once *offline*. Because a structured NL expresses a finite set of TL specifications, we can enumerate finitely many possible canonicalized decomposition trees that each express all possible full specifications (modulo variables names and boolean expressions). Then, deriving a proxy for a sub-requirement in such a tree captures common behavior among all full specifications that could ever contain it. We conduct this derivation offline, and then, to produce a proxy for a structured sub-requirement in a node in any decomposition tree, SSG instantiates an offline-generated proxy by replacing its variable names and Boolean expressions with those used in the structured sub-requirement.

6 Balanced Distinguishing Trace Generation

Recall from §3.3 that ARTEMIS introduces Balanced Distinguishing Trace Generation (BDTG), to reduce the worst-case number of distinguishing traces from $O(n)$ [20] to $O(log(n))$. This section details how BDTG achieves this efficiency.

Given a set of candidate TL specifications $S = \{\phi_1, \phi_2, ..., \phi_n\}$, BDTG generates distinguishing traces that minimize the worst-case number of traces users need to inspect. For a trace t, let $n_{allow}(t)$ ($n_{disallow}(t)$) denote the number of candidates that (dis-)allow t. If the user accepts trace t, then $n_{disallow}(t)$ candidates are eliminated. Else, $n_{allow}(t)$ candidates are eliminated. Intuitively, BDTG aims to generate *balanced* traces that eliminate half of the remaining candidates regardless of user response, maximizing worst-case number eliminations per trace. Concretely, its objective is to produce balanced trace t^* where:

$$\max_{t^* \in \phi_1 \vee \phi_2 \vee \cdots \vee \phi_n} (\min(n_{\text{allow}}(t^*), n_{\text{disallow}}(t^*))) \tag{3}$$

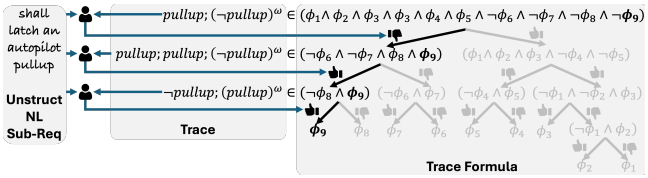


Figure 5: Illustrates BDTG converging to the ninth candidate "at next point pullup" of node C (Fig. 2). Given TL specifications $\phi_1, ..., \phi_9$ representing proxies for each of the candidates, BDTG produces distinguishing trace between them so that half of the remaining candidates are pruned per user query.

Fig. 5 illustrates Balanced Distinguishing Trace Generation (BDTG) for node C of Fig. 2. Given nine candidate proxies $\phi_1 \ldots \phi_9$ for node C's structured sub-requirements, BDTG produces distinguishing traces that prune half the remaining candidates with each user query, converging to the ninth candidate in just three queries.

One could exhaustively enumerate all $O(2^n)$ possible combinations of TL formulas and their negations to find balanced distinguishing trace t^*. However this is likely infeasible for large n.

ARTEMIS's BDTG addresses this scalability challenge in three ways. First, as with Sub-Specification Generation (§5), BDTG is applied to each node individually. Given a translation decomposition with N nodes and K structured NL sub-requirements per node, the search space size is $O(2^K)$ for each of the N nodes, which is significantly smaller than considering all possible $O(2^{(K^N)})$ full formulas. Second, BDTG introduces a guided search procedure (§6.1) that avoids redundant model checker queries by tracking partial order relations (subset, mutual exclusion) between candidates. Finally, BDTG introduces optional configurable hyperparameter d (§6.1) to trade off global optimality of the objective (Eq. 3) for scalability. If d is set, ARTEMIS partitions all candidates into groups of size $\leq d$ and exhaustively searches a group for a balanced trace. Our evaluation (§7.3) shows that even with $d < n$, ARTEMIS achieves results similar to the optimal number of queries to the user.

6.1 Guided Search by Partial Order Relations

Given candidate set S, BDTG first computes partial order relations between pairs of candidates:

- ϕ_i is a subset of ϕ_j if every trace satisfying ϕ_i also satisfies ϕ_j.
- ϕ_i and ϕ_j are mutually exclusive if no trace satisfies both.

Consider an example set of n candidates where $\phi_i \subseteq \phi_{i+1}$ for $i \in \{1, 2, ..., n-1\}$. These relations imply that any trace formula with $\phi_i \wedge \neg\phi_{i+1}$ is unsatisfiable, reducing the search space from 2^n to $n+1$ possibly satisfiable trace formulas. In Figure 5, partial order relations among node C's nine candidates reduce the search space significantly. E.g., candidates representing"immediately pullup" and "eventually pullup" have a subset relationship, allowing BDTG to prune unsatisfiable combinations.

After computing partial order relations, BDTG partitions the candidates into groups of size $\leq d$. To minimize redundant model checker queries, candidates are grouped to maximize the number of partial order relations within each group. BDTG then searches for a satisfiable trace formula per group, given that the trace formulas of each group must be satisfiable in conjunction.

Given constant d ($d = 10$ in our evaluation) and input size n, BDTG's time complexity is $O(n^2 + \frac{n}{d}2^d) = O(n^2)$ (significantly lower than exhaustive search $O(2^n)$). Combined with Sub-Specification Generation, this yields $O(K^2)$ search space per node for decomposition trees with N nodes and K sub-requirements each.

6.2 Manual Effort Analysis

When hyperparameter $d \geq n$, BDTG is guaranteed to produce a trace that achieves the maximum number of eliminations for any user response (optimal for Eq. 3). The maximum achievable eliminations vary by candidate set. For example, if all candidates are mutually exclusive (no shared traces), then the maximum achievable eliminations for any trace and any user response is one.

Given n candidates and hyperparameter $d \geq n$, we show that BDTG converges to a single candidate in $O(\log(n))$ queries to the user for all but a vanishing fraction of possible input sets of candidates. Observe that most of the trace formulas among all 2^n possible eliminate nearly half the number of candidates. Concretely, the fraction of all possible trace formulas that eliminate at least $(\frac{1}{2} - \delta)n$ candidates approaches one as n grows:

$$\frac{2\sum_{k=\lceil(\frac{1}{2}-\delta)n\rceil}^{n/2-1}\binom{n}{k} + \binom{n}{n/2}}{2^n} \rightarrow 1 \text{ as } n \rightarrow \infty$$

where δ is an arbitrarily small positive constant ($0 < \delta << 0.5$). It follows that, as n increases, the fraction of all possible input candidate sets exhibiting a satisfiable trace formula that eliminates at least $(\frac{1}{2} - \delta)n$ candidates approaches one. Thus, BDTG converges to a single candidate in $O(log_{\frac{2}{1+2\delta}}(n)) = O(log(n))$ traces for all but a vanishing fraction of inputs as n grows.

Given N nodes and up to K structured NL sub-requirements per node, Sub-Specification Generation and BDTG together converge to one candidate specification in $O(Nlog(K))$ traces in most cases (much lower than all possible TL specifications $O(K^N)$). In Figure 5, node C has nine candidates. Without BDTG, prior work [20] would require up to 8 traces ($O(n)$). With BDTG, ARTEMIS converges in 3 traces, demonstrating its practical benefits.

7 Evaluation

We investigate the following two research questions:

- **RQ1**: How does ARTEMIS's LLM-based translation approach compare to prior work w.r.t. output specification accuracy?
- **RQ2**: How does ARTEMIS's validation approach, Sub-Specification Generation and Balanced Distinguishing Trace Generation, compare to prior work w.r.t. manual effort?

7.1 Experiment Setup

Implementation and Hardware Setup. We implement ARTEMIS in ~5K lines of Python, using open-source LTL model checkers Spot [12] and nuXmv [5]. For all LTL-based approaches (§7.3), Spot is queried first, falling back to nuXmv after a 1-second timeout. We configured Spot and nuXmv to generate traces up to length 20. Experiments were run on a MacBook Pro (M3 Pro, 18 GB RAM). The LLM prompts use chain-of-thought [53] (CoT) prompting, which instructs the LLM to generate intermediate reasoning that leads to a final answer. The prompt contains the input requirement and the structured NL grammar, and instructs the LLM to translate the input

Benchmark	Vent. (-H)	Robo. (-H)	LMCPS (-H)	DeepSTL (-H)	Thales (-H)
# requirements	121 (89)	46 (21)	15 (4)	14 (5)	22 (15)
# plausible range	1–64 (1–64)	1–10 (1–6)	1–20 (1–20)	1 (1)	1 (1)

Table 1: Summary of evaluated benchmarks.

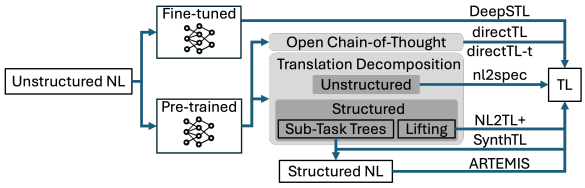


Figure 6: ARTEMIS vs. baseline prompting strategies.

to structured NL by identifying unstructured sub-requirements and mapping each to k structured sub-requirement candidates (§4). All code, benchmarks, results, baselines, and example prompts can be found in our artifact [42].

Benchmarks. We evaluate ARTEMIS and prior approaches on real-world NL requirements from diverse domains, with each benchmark containing requirements and corresponding expert-curated LTL specifications. Due to NL ambiguity, some requirements have multiple plausible specifications vetted by a domain-expert. Table 1 summarizes the size and number of plausible specifications per benchmark (§2.3). We exclude synthetic benchmarks and those that are unrepresentative of real-world requirements (e.g., [9, 10]).

The **Ventilator** benchmark [16] contains 121 NL requirements for a safety-critical ventilator system. The **LMCPS** benchmark [14], created by Lockheed Martin, contains 15 NL aviation requirements for an autopilot FSM and a control loop regulator. The **Robotics** benchmark [50] comprises 46 NL requirements from the nuclear domain, focused on robot-human teamwork explainability. All three sets were previously formalized in FRETish [16, 40, 50].

The **DeepSTL** benchmark [28] consists of 14 NL requirements paired to STL expressions. The requirements originate from diverse domains, including driving, robotics, and electronics, and were previously used to evaluate the accuracy of NLP-based specification generation. Two TL experts approximated the LTL from STL by discretizing the time intervals and representing real-valued predicates as atomic propositions, and then expressed them in FRETish.

The **Thales** benchmark [2] contains 22 NL requirements of an industrial indoor positioning system with interactive mobile ad-hoc devices. Prior work formalized these requirements in PSP [2].

For benchmarks previously formalized in FRETish (Ventilator [16], Robotics [50], LMCPS [14]), we asked three domain experts to enumerate additional plausible FRETish specifications per requirement by considering as many reasonable interpretations as possible. Ventilator-H, Robotics-H, LMCPS-H, DeepSTL-H, and Thales-H denote subsets with plausible specifications larger than five symbols (operators or atomic propositions), enabling a comparison of approaches on more complex specifications w.r.t. formula size.

Structured NLs. To evaluate the efficacy of ARTEMIS's LLM Translation to Structured NL, we use two state-of-the-art structured NLs for its IR: **FRETish** (§2.2) and **PSP** [2].

PSP extends SPS [13] with 1) a structured NL grammar for expressing requirements, 2) timed variants of SPS patterns, and 3) new patterns. Each PSP requirement includes a *scope* (defining the trace interval where the requirement applies, e.g., *after* or *before* an event) and a *pattern*, expressed as either an *occurrence* (e.g., existence/absence of an event) or *ordering* (e.g., a response to an event).

ARTEMIS produces proxies (§5) for FRETish/PSP identically, using Eq. 1 for standard proxies and Eq. 2 to upgrade them to abstract proxies. For each benchmark, ARTEMIS uses the same structured NL IR as that of its plausible specifications. FRETish for Ventilator, LMCPS, Robotics, and DeepSTL; PSP for Thales.

LLMs. We evaluate ARTEMIS and baselines using OpenAI's gpt-4.1 (**GPT**) [1] and Google's gemini-2.5-flash (**gemini**) [21].

7.2 Evaluating Translation Accuracy

We compare the translation accuracy of ARTEMIS's LLM Translation to Structured NL to a set of baselines (RQ1), all of which translate unstructured NL requirements to TL specifications using LLM prompting. Baselines use either pre-trained [8, 10, 34, 38, 41] or fine-tuned [6, 28, 37] LLMs, the latter of which have undergone additional training on curated datasets containing NL-TL pairs.

Baselines Prompting Pre-Trained LLMs. Baselines that prompt pre-trained LLMs use CoT prompting [8, 10, 34, 38, 41], which can be *open* (freely generated reasoning) or *guided* (specialized reasoning).

We implement two open CoT baselines, **directTL** and **directTL-t** (Fig. 6), which leverage prompts designed after existing open CoT approaches [8, 34]. Both instruct an LLM to translate an unstructured NL requirement to a TL specification and generate an explanation. In directTL-t, the prompt also provides all TL templates expressible by the structured NL (i.e., either FRETish or PSP) and asks the LLM to select one. This makes directTL-t analogous to ARTEMIS, with a key distinction: directTL-t provides a set of TL templates, whereas ARTEMIS provides a structured NL grammar. Thus, the directTL-t baseline isolates the effect of using LTL templates without a structured NL grammar.

Guided CoT baselines use translation decomposition (Fig. 6) [6, 10, 37, 38, 41]. nl2spec [10] and CoT-TL [38] use unstructured translation decomposition (§2.3). CoT-TL adds external context via Semantic Role Labeling (SRL) [46], while nl2spec and ARTEMIS do not, making **nl2spec** a more suitable baseline for isolating the effect of structured NL IRs on translation accuracy.

We categorize approaches using structured translation decomposition (§2.3) as using *lifting* [19, 37] or *sub-task trees* [41] (Fig. 6).

Lifting translates an NL requirement to a TL specification in three steps. First, a pre-trained LLM replaces domain-specific phrases with placeholder symbols, producing lifted NL. Next, an LLM converts the lifted NL into lifted TL. Finally, the LLM maps each placeholder to an atomic proposition, yielding the final TL specification. NL2TL [6] and Lang2LTL [37] are the only tools using lifting. Both assume each placeholder maps to a single atomic proposition, whereas in our benchmarks, placeholders map to Boolean expressions over multiple atomic propositions. Lang2LTL's embedding cannot support mapping to Boolean expressions; thus, to produce a representative baseline, we adapt NL2TL into **NL2TL+** by adding a prompt to map placeholders to Boolean expressions.

Like ARTEMIS, **SynthTL** (§2.3) uses LLMs to decompose a translation task into a logical tree of sub-tasks.

Baselines Prompting Fine-Tuned LLMs. Fine-tuning an LLM requires a large in-domain dataset. Since collecting one demands

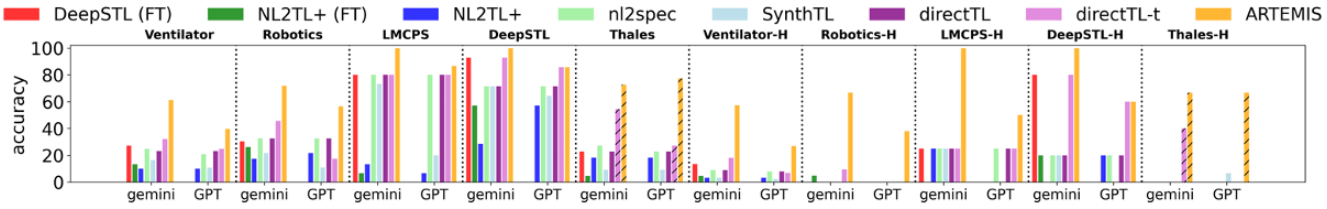


Figure 7: Specification accuracy of existing LLM prompting vs. ARTEMIS. Stripes in directTL-t/ARTEMIS denote use of PSP.

substantial manual effort and expertise, baselines using fine-tuned LLMs [6, 28, 37] rely on manually curated unstructured NL-TL pairs to support synthetic generation of supervised fine-tuning data.

For our baselines using fine-tuned LLMs, we fine-tune gemini [21] on the DeepSTL dataset [28], since it comprehensively covers commonly used TL patterns across many domains (whereas [6, 37] derive training data from specific domains). The baseline **DeepSTL (FT)** uses the fine-tuned LLM to translate unstructured NL directly to TL without auxiliary CoT (Fig. 6). Since NL2TL [6] uses a fine-tuned LLM for lifted NL to lifted TL translation, we also evaluate **NL2TL+ (FT)**, which uses a fine-tuned LLM for this step.

In all baselines and ARTEMIS, each output is checked for syntax validity. If invalid, the syntax error is appended to the prompt and the LLM is asked to fix it. Three failed attempts clear the prompt and trigger a retry until producing $k = 10$ syntactically valid outputs.

All translation decomposition baselines (nl2spec, SynthTL, NL2TL+) use few-shot prompts with input–output–reasoning examples to demonstrate decomposition; for consistency, we use each baseline's original few-shot examples. To isolate the impact of IRs and prompting strategies, we conservatively configure ARTEMIS, directTL, and directTL-t to use zero-shot prompting (no examples).

For all benchmarks except Robotics, all approaches' prompts contain a component-wise list of atomic propositions to use, following common practice to enable checking generated/plausible specification equivalence [6, 10, 37, 41]. For Robotics, a requirement-wise list is used instead, since its components contain synonymous atomic propositions, making component-wise lists redundant.

Orthogonal Translation Optimizations Other work on LLM-based requirement to specification translation proposes strategies to optimize an LLM's outputs through including external context (i.e., retrieval augmented generation (RAG) [15, 55] or SRL [38, 46]) or through refining LLM outputs via LLM self-refinement [30, 33, 35]. Optimizations using external context require a large database of (rare) in-domain context/examples to be effective and may not improve accuracy [36]. Self-refinement relies on feedback from external oracles/tools to be effective, and without them, it is unlikely to improve accuracy [29, 31]. Since we aim to measure the impact of using different IRs, we omit such optimizations. We expect them to benefit ARTEMIS similarly to how they benefit prior work.

Metrics. To measure **accuracy**, each approach independently generates $k = 10$ candidates per requirement, and we report the percentage of requirements per benchmark for which at least one plausible specification is produced. For all approaches, plausibility is assessed by LTL equivalence to a plausible LTL specification.

Results. Fig. 7 shows the accuracy of ARTEMIS and baselines on each of the benchmarks. Using gemini/GPT, ARTEMIS achieves 1.9X/1.6X, 1.57X/1.73X, 1.25X/1.08X, 1.08X/1X, and 1.33X/2.83X higher accuracy compared to the best baseline on the Ventilator, Robotics, LMCPS, DeepSTL, and Thales benchmarks, respectively. Across all benchmarks, ARTEMIS achieves 1.57X/1.43X higher accuracy compared to the best baseline using gemini/GPT, respectively. Observe that the accuracy difference between ARTEMIS and the baselines is larger on more complex requirements, i.e., all those with "-H" appended. In fact, many baselines achieve 0% accuracy. Using gemini/GPT, ARTEMIS achieves 3.19X/3.43X, 7X/-, 4X/2X, 1.25X/1X, and 1.67X/10X higher accuracy compared to the best baseline on the Ventilator-H, Robotics-H, LMCPS-H, DeepSTL-H, and Thales-H benchmarks, respectively. The "-" indicates that the best baseline achieves 0% accuracy on the Robotics-H benchmark with GPT, whereas ARTEMIS achieves 38% accuracy.

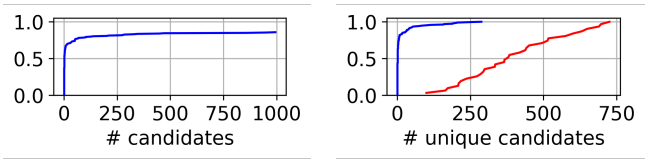
We identified 101 requirements with plausible specifications that SynthTL cannot decompose, as they involve FRETish `scope`, `condition`, or PSP scope fields that cannot be expressed as subformulas. On these requirements, SynthTL achieves 0%/0% accuracy and ARTEMIS achieves 61.39%/26.73% using gemini/GPT, demonstrating SynthTL's decomposition expressiveness limitation (§4).

7.3 Evaluating Manual Validation Effort

To evaluate manual effort in producing a plausible specification (RQ2), we measure how many candidate specifications an LLM must generate to yield one. We then compare effort spent in converging to a plausible specification using ARTEMIS and prior work.

Number of Candidates to Produce Plausible Specifications. We configure ARTEMIS to query gemini to generate candidate specifications until it produces one plausible specification, capped at 1000 candidates per requirement. To elicit unique candidates, each LLM prompt contains all previously generated outputs and instructs the LLM to generate the top $k = 50$ most likely specifications that are different from the previous ones. For practical reasons we use $k = 50$: setting k too small makes generating 1000 candidates extremely costly, while setting k too large exceeds API limits on the maximum number of output tokens per query.

Results. Fig. 8a (Fig. 8b) shows the cumulative distribution function, or CDF, of the number of (unique) candidates generated until producing a plausible specification, among all requirements in all benchmarks. A plausible specification was not produced for 14.22% of all requirements, indicating these requirements need more than 1000 candidates. The maximum (99-th percentile) number of candidates to produce a plausible specification is 994 (854). This result demonstrates that *many* candidates must be generated per requirement to ensure at least one is plausible. The maximum (99-th percentile) number of unique candidates is 726 (690). Thus,



(a) Candidates needed to produce a plausible specification.

(b) Unique candidates until producing a plausible specification.

Figure 8: CDFs of candidates. Blue (red) distribution indicates success (failure) within 1000 candidates.

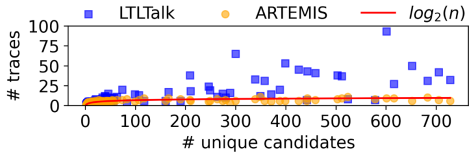


Figure 9: Traces inspected to produce a plausible output.

distinguishing trace generation must handle upwards of hundreds of specifications—many more than considered in prior work [20].

Baseline Comparison. We compare manual effort required by state-of-the-art LTLTalk (§2.3) and ARTEMIS to converge to a plausible specification using distinguishing traces. To simulate a user, traces are accepted/rejected according to a plausible specification produced by an expert. We use ARTEMIS (due to its higher translation accuracy, §7.2) to generate up to 1000 candidates per requirement, stopping early upon producing a plausible candidate (Fig. 8). We augment candidates for each requirement with expert-curated specifications to ensure at least one plausible specification per requirement. Both ARTEMIS and LTLTalk are given the same set of candidates; ARTEMIS expresses them as a decomposition tree and LTLTalk expresses them as a set of LTL specifications. For all input requirements, ARTEMIS produces a tree with less than four nodes, and all unstructured sub-requirements are substrings of the input. Since Decomposition Validation (§4) requires only intuitive reasoning over NL, this effort is negligible compared to trace inspection. We set ARTEMIS's grouping hyperparameter to $d = 10$ (§6).

Metrics. For each requirement in each benchmark, we collect: the total **number of traces** to converge to a plausible specification, the **number of variables** within each trace, the **trace lengths**, and **runtime overheads** in generating traces. We calculate trace length as the number of steps in its prefix and cycle.

Manual Validation Effort. Fig. 9 compares the number of traces inspected vs. number of unique candidates using ARTEMIS and LTLTalk for all requirements in each benchmark. Observe that with n candidate specifications, the number of traces with ARTEMIS follows a $log_2(n)$ trend (§6.2). The largest number of traces inspected for any requirement using LTLTalk and ARTEMIS is 93 and 11, respectively (8.45X lower for ARTEMIS). ARTEMIS converged to a plausible specification with up to 10.83X fewer traces (1.74X on average) and up to 6X (1.40X on average) fewer variables per trace than LTLTalk (3.33 variables per trace on average for ARTEMIS). As the number of candidates increases, the difference in manual effort required by LTLTalk and ARTEMIS becomes greater. For requirements with more than 10 candidates, ARTEMIS requires 2.52X fewer traces on average. The trace lengths for traces produced by LTLTalk and ARTEMIS are 2.61 (up to 20) and 3.1 (up to 19). Evidently, the state-of-the-art can require an impractical amount of effort for a single requirement (up to 93 traces). In contrast, even for many candidates, ARTEMIS enables low effort, practical use of generate-and-validate (up to 11 traces).

Method	Mean ± Std. Dev.	Max Time	Min Time
LTLTalk	7.08 ± 27.22	273.27	0.08
ARTEMIS	18.89 ± 121.76	2437.30	0.36

Table 2: Execution times for trace generation.

Trace Generation Runtime Overhead. Table 2 shows the average and maximum distinguishing trace generation runtimes of LTLTalk and ARTEMIS. ARTEMIS on average generates traces in 18.89 seconds, demonstrating its practicality. ARTEMIS's trace generation runtime could be further decreased by lowering d (§6), but may result in requiring the user to inspect more traces. Although ARTEMIS's trace generation may exhibit higher runtime overhead than LTLTalk, trace generation is automated, and ARTEMIS achieves significantly lower manual effort.

Proxy Generation Runtime Overhead. For Sub-Specification Generation (§5), ARTEMIS generates proxies only once for a structured NL (§5.2). ARTEMIS generates proxies for all FRETish templates in 134 seconds and for all PSP templates in 231 seconds.

7.4 Threats to Validity

Our evaluation uses five real-world benchmarks from diverse domains, covering a broad range of LTL specifications. However, they may not be representative of every specification used in practice.

Each requirement in our evaluation has a plausible specification expressible in structured NL. While prior work [2, 13, 24] shows most real-world requirements can be captured by a finite set of specification patterns (observed to be over 90% [13]), some may fall outside the structured NL's expressiveness. However, such cases are likely rare, and the structured NL can be extended as needed.

For distinguishing trace generation, we set all approaches to generate traces up to length 20. Although some specifications require longer traces, this is rare: all approaches have an average trace length below four across all requirements.

8 Conclusion

While LLMs show promise for automating specification generation from NL, prior work requires substantial expertise and manual validation effort, limiting practicality. ARTEMIS introduces a new generate-and-validate approach that reduces both effort and expertise needed, achieving 1.57X higher accuracy and up to 10.38X lower effort on real-world requirements compared to state-of-the-art baselines, enabling practical adoption.

Acknowledgments

We thank Tom Pressburger, Kelly Ho and, Jessica Phelan for their work on early prototypes, and we are grateful to the anonymous reviewers for their constructive feedback. Anastasia and Andreas were supported in part by NASA contract 80ARC020D0010. Daniel and Caroline were supported in part by the National Science Foundation (NSF), under awards 2153936 and 2236855 (CAREER), and also gratefully acknowledge a gift from Google.

References

[1] OpenAI (2023). 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[2] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. 2015. Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Transactions on Software Engineering* 41, 7 (2015), 620–638. doi:10.1109/TSE.2015.2398877

[3] Mishap Investigation Board. 1999. *Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999.* Technical Report. Report.

[4] Alberto Camacho and Sheila A. McIlraith. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. *Proceedings of the International Conference on Automated Planning and Scheduling* 29, 1 (Jul. 2019), 621–630. doi:10.1609/icaps.v29i1.3529

[5] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 334–342.

[6] Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. 2023. NL2TL: Transforming Natural Languages to Temporal Logics using Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model checking*. MIT Press, Cambridge, MA, USA.

[8] Itay Cohen and Doron Peled. 2025. End-to-End AI Generated Runtime Verification from Natural Language Specification. In *Bridging the Gap Between AI and Reality*, Bernhard Steffen (Ed.). Springer Nature Switzerland, Cham, 362–384.

[9] Riccardo Coltrinari. 2023. NL-to-LTL-Synthetic-Dataset. https://huggingface.co/datasets/cRick/NL-to-LTL-Synthetic-Dataset.

[10] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 383–396.

[11] RTCA DO-178C. 2011. Software Considerations in Airborne Systems and Equipment Certification. (2011).

[12] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022. From Spot 2.0 to Spot 2.10: What's New?. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22) (Lecture Notes in Computer Science, Vol. 13372)*. Springer, 174–187. doi:10.1007/978-3-031-13188-2_9

[13] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 411–420. doi:10.1145/302405.302672

[14] Chris Elliott. 2015. On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works. In *Safe & Secure Systems and Software Symposium (S5)*. 9–11.

[15] Yue Fang, Zhi Jin, Jie An, Hongshen Chen, Xiaohong Chen, and Naijun Zhan. 2025. Enhancing Transformation from Natural Language to Signal Temporal Logic Using LLMs with Diverse External Knowledge. arXiv:2505.20658 [cs.CL] https://arxiv.org/abs/2505.20658

[16] Marie Farrell, Matt Luckcuck, Rosemary Monahan, Conor Reynolds, and Oisín Sheridan. 2024. FRETting and Formal Modelling: A Mechanical Lung Ventilator. In *Rigorous State-Based Methods*, Silvia Bonfanti, Angelo Gargantini, Michael Leuschel, Elvinia Riccobene, and Patrizia Scandurra (Eds.). Springer Nature Switzerland, Cham, 360–383.

[17] Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. 2017. Pure: A dataset of public requirements documents. In *2017 IEEE 25th international requirements engineering conference (RE)*. IEEE, 502–505.

[18] Nicole Fronda and Houssam Abbas. 2022. Differentiable Inference of Temporal Logic Formulas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4193–4204. doi:10.1109/TCAD.2022.3197506

[19] Francesco Fuggitti and Tathagata Chakraborti. 2023. NL2LTL - a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 1999, 3 pages. doi:10.1609/aaai.v37i13.27068

[20] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive synthesis of temporal specifications from examples and natural language. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 201 (nov 2020), 26 pages. doi:10.1145/3428269

[21] Google Gemini Team. 2025. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[22] Enrico Ghiorzi, Michele Colledanchise, Gianluca Piquet, Stefano Bernagozzi, Armando Tacchella, and Lorenzo Natale. 2023. Learning Linear Temporal Properties for Autonomous Robotic Systems. *IEEE Robotics and Automation Letters* 8, 5 (2023), 2930–2937. doi:10.1109/LRA.2023.3263368

[23] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2016. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings 8*. Springer, 41–46.

[24] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2021. Automated formalization of structured natural language requirements. *Information and Software Technology* 137 (2021), 106590. doi:10.1016/j.infsof.2021.106590

[25] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. 2011. Synthesis of AMBA AHB from formal specification: a case study. *International Journal on Software Tools for Technology Transfer* 15 (2011), 585 – 601. https://api.semanticscholar.org/CorpusID:15838863

[26] Lars Grunske. 2008. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/1368088.1368094

[27] Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal Specifications from Natural Language. arXiv:2206.01962 [cs.SE]

[28] Jie He, Ezio Bartocci, Dejan Ničković, Haris Isakovic, and Radu Grosu. 2022. DeepSTL: from english requirements to signal temporal logic. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 610–622. doi:10.1145/3510003.3510171

[29] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models Cannot Self-Correct Reasoning Yet. In *Proceedings of the 12th International Conference on Learning Representations*. https://arxiv.org/abs/2310.01798

[30] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards Mitigating LLM Hallucination via Self Reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 1827–1843. doi:10.18653/v1/2023.findings-emnlp.123

[31] Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. *Transactions of the Association for Computational Linguistics* 12 (2024), 1417–1440. doi:10.1162/tacl_a_00713

[32] Sascha Konrad and Betty H. C. Cheng. 2006. Automated Analysis of Natural Language Properties for UML Models. In *Satellite Events at the MoDELS 2005 Conference*, Jean-Michel Bruel (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 48–57.

[33] Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. 2025. Safeguarding Mobile GUI Agent via Logic-based Action Verification. arXiv:2503.18492 [cs.HC] https://arxiv.org/abs/2503.18492

[34] Hui Li, Zhen Dong, Siao Wang, Hui Zhang, Liwei Shen, Xin Peng, and Dongdong She. 2025. Extracting Formal Specifications from Documents Using LLMs for Automated Testing. arXiv:2504.01294 [cs.SE] https://arxiv.org/abs/2504.01294

[35] Junle Li, Meiqi Tian, and Bingzhuo Zhong. 2025. Automatic Generation of Safety-compliant Linear Temporal Logic via Large Language Model: A Self-supervised Framework. arXiv:2503.15840 [cs.LO] https://arxiv.org/abs/2503.15840

[36] Jingyu Liu, Jiaen Lin, and Yong Liu. 2024. How Much Can RAG Help the Reasoning of LLM? arXiv:2410.02338 [cs.CL] https://arxiv.org/abs/2410.02338

[37] Jason Xinyu Liu, Ziyi Yang, Benjamin Schornstein, Sam Liang, Ifrah Idrees, Stefanie Tellex, and Ankit Shah. 2022. Lang2LTL: Translating Natural Language Commands to Temporal Specification with Large Language Models. In *Workshop on Language and Robotics at CoRL 2022*. https://openreview.net/forum?id=VxfjGZzrdn

[38] Kumar Manas, Stefan Zwicklbauer, and Adrian Paschke. 2024. CoT-TL: Low-Resource Temporal Knowledge Representation of Planning Instructions Using Chain-of-Thought Reasoning. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 9636–9643. doi:10.1109/IROS58592.2024.10801817

[39] Alistair Mavin. 2012. Listen, Then Use EARS. *IEEE Software* 29, 2 (2012), 17–18. doi:10.1109/MS.2012.36

[40] Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Tom Pressburger, Mohammad Hejase, Pierre-Loic Garoche, and Johann Schumann. 2020. The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. In *Proceedings of the 2020 28th IEEE International Requirements Engineering Conference*.

[41] Daniel Mendoza, Christopher Hahn, and Caroline Trippel. 2024. Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2024*. 119.

[42] Daniel Mendoza, Anastasia Mavridou, Andreas Katis, and Caroline Trippel. 2026. ARTEMIS Artifact. https://github.com/dmmendo/ARTEMIS.

[43] Sara Mohammadinejad, Sheryl Paul, Yuan Xia, Vidisha Kudalkar, Jesse Thomason, and Jyotirmoy V. Deshmukh. 2025. Systematic Translation from Natural Language Robot Task Descriptions to STL. In *Bridging the Gap Between AI and Reality*, Bernhard Steffen (Ed.). Springer Nature Switzerland, Cham, 259–276.

[44] Anmol Nayak, Hari Prasad Timmapathini, Vidhya Murali, Karthikeyan Ponnalagu, Vijendran Gopalan Venkoparao, and Amalinda Post. 2022. Req2Spec: Transforming Software Requirements into Formal Specifications Using Natural Language Processing. In *Requirements Engineering: Foundation for Software Quality*, Vincenzo Gervasi and Andreas Vogelsang (Eds.). Springer International Publishing, Cham, 87–95.

[45] Daniel Neider and Ivan Gavran. 2018. Learning Linear Temporal Properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–10. doi:10.23919/FMCAD.2018.8603016

[46] Martha Palmer, Claire Bonial, and Jena Hwang. 2017. 315VerbNet: Capturing English Verb Behavior, Meaning, and Usage. In *The Oxford Handbook of Cognitive Science*. Oxford University Press. arXiv:https://academic.oup.com/book/0/chapter/295177378/chapter-ag-pdf/44511408/book_34641_section_295177378.ag.pdf doi:10.1093/oxfordhb/9780199842193.013.15

[47] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. doi:10.1109/SFCS.1977.32

[48] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. 2022. Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 263–280.

[49] Kristin Yvonne Rozier. 2016. Specification: The Biggest Bottleneck in Formal Methods and Autonomy. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 8–26.

[50] Hazel Taylor, Anastasia Mavridou, Marie Farrell, and Louise Dennis. 2025. Explainability Pattern Specifications for Human-Robot Teamwork. In *IEEE International Conference on Engineering Reliable Autonomous Systems (ERAS-25)*.

[51] Haoxiang Tian, Guoquan Wu, Jiren Yan, Yan Jiang, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2023. Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.

[52] Christopher Wang, Candace Ross, Yen-Ling Kuo, Boris Katz, and Andrei Barbu. 2021. Learning a natural-language to LTL executable semantic parser for grounded robotics. In *Proceedings of the 2020 Conference on Robot Learning (Proceedings of Machine Learning Research, Vol. 155)*, Jens Kober, Fabio Ramos, and Claire Tomlin (Eds.). PMLR, 1706–1718. https://proceedings.mlr.press/v155/wang21g.html

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[54] Ran Wei, Simon Foster, Haitao Mei, Fang Yan, Ruizhe Yang, Ibrahim Habli, Colin O'Halloran, Nick Tudor, Tim Kelly, and Yakoub Nemouchi. 2024. ACCESS: Assurance Case Centric Engineering of Safety–critical Systems. *J. Syst. Softw.* (2024). doi:10.1016/j.jss.2024.112034

[55] Yilongfei Xu, Jincao Feng, and Weikai Miao. 2024. Learning from Failures: Translation of Natural Language Requirements into Linear Temporal Logic with Large Language Models. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. 204–215. doi:10.1109/QRS62785.2024.00029

[56] Changjian Zhang, Parv Kapoor, Ian Dardik, Leyi Cui, Romulo Meira-Goes, David Garlan, and Eunsuk Kang. 2025. Constrained LTL Specification Learning from Examples.

[57] Rim Zrelli, Henrique Amaral Misson, Maroua Ben Attia, Felipe Gohring de Magalhães, Abdo Shabah, and Gabriela Nicolescu. 2024. Natural2ctl: A dataset for natural language requirements and their ctl formal equivalents. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 205–216.