ClarifySTL: An Interactive LLM Agent Framework for STL Transformation through Requirements Clarification

YUE FANG, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China
ZHI JIN*, School of Computer Science, Wuhan University, China and Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China
JIE AN*, National Key Laboratory of Space Integrated Information System, Institute of Software Chinese Academy of Sciences, China
JIA LI, School of Computer Science, Wuhan University, China
HONGSHEN CHEN, JD.com, China
XIAOHONG CHEN, East China Normal University, China
NAIJUN ZHAN, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China

Signal Temporal Logic (STL) is a formal language for specifying real-time behaviors of cyber-physical systems (CPS). Automatically transforming natural language requirements into STL specifications has received growing attention. Recent efforts leveraging large language models (LLMs) have demonstrated impressive performance, but some natural language requirements in practice contain vague or ambiguous information, which remains challenging for LLMs to handle. To address these challenges, we propose ClarifySTL, an interactive LLM-agent framework that enhances STL transformation through requirements clarification. ClarifySTL first detects vague expressions that indicate underspecified information in a requirement. If any vagueness is detected, it generates targeted clarification queries to guide users in supplementing the requirement until all necessary details are provided. Subsequently, if ClarifySTL detects ambiguities, it formulates focused ambiguity clarification queries and updates the requirements based on user feedback until all ambiguities are resolved. Finally, the requirements with vagueness and ambiguity clarified are transformed into STL specifications using LLMs. This interactive framework ensures that the resulting STL formulas faithfully capture user intent while reducing the burden on the user. We evaluate ClarifySTL on the representative benchmarks DeepSTL and STL-DivEn, as well as our newly introduced AmbiEval benchmark, which is specifically designed to assess the performance of the agents in handling vagueness and ambiguity, including both detection and query generation. The experimental results show that ClarifySTL outperforms the state-of-the-art model by 13.92% and 12.08% in Formula Accuracy and Template Accuracy on the DeepSTL benchmark, and by 12.57%

*Corresponding Author.

Authors' Contact Information: Yue Fang, y.fang@stu.pku.edu.cn, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, Beijing, China; Zhi Jin, zhijin@pku.edu.cn, School of Computer Science, Wuhan University, Wuhan, China and Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, Beijing, China; Jie An, anjie@iscas.ac.cn, National Key Laboratory of Space Integrated Information System, Institute of Software Chinese Academy of Sciences, Beijing, China; Jia Li, jia.li@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, China; Hongshen Chen, ac@chenhongshen.com, JD.com, Beijing, China; Xiaohong Chen, xhchen@sei.ecnh.edu.cn, East China Normal University, Shanghai, China; Naijun Zhan, njzhan@pku.edu.cn, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, Beijing, China.

ACM 1557-7392/2025/8-ART
https://doi.org/XXXXXXX.XXXXXXX

and 12.99% in the same metrics on the STL-DivEn benchmark respectively, and achieves an average 90.9% detection accuracy for vagueness and ambiguity. Human evaluations further validate the effectiveness of our framework, which clarifies 93.8% of defective requirements and generates effective queries for vagueness and ambiguity at rates of 93.3% and 94.3%, respectively.

CCS Concepts: • **Software and its engineering** → **Specification languages**; **Design languages**.

Additional Key Words and Phrases: Signal Temporal Logic, Requirements Clarification, LLM-agent

ACM Reference Format:
Yue Fang, Zhi Jin, Jie An, Jia Li, Hongshen Chen, Xiaohong Chen, and Naijun Zhan. 2025. ClarifySTL: An Interactive LLM Agent Framework for STL Transformation through Requirements Clarification. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (August 2025), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Temporal Logic (TL) [46] provides a powerful tool for precisely expressing the dynamic behavior of systems in the analysis and verification of cyber-physical systems (CPS). Signal Temporal Logic (STL) [38], an extension of TL, introduces real-time and real-valued constraints, enabling the description of not only discrete temporal events but also continuous-time and real-valued dynamic changes. With these capabilities, STL has been widely used in safety-critical CPS such as autonomous driving and robotic control, helping system designers precisely specify temporal requirements of system behaviors and drawing growing attention from both industry and academia [24, 37, 51, 57]. However, most system requirements with temporal constraints remain documented in informal natural language by domain experts. Accurately transforming such requirements into formal STL specifications poses a major challenge to the broader application of STL in practical CPS development and verification.

Manually writing precise STL formulas is a laborious task for domain experts, as the process is both time-consuming and error-prone. Therefore, many studies [5, 36, 39] have attempted to develop automated methods to transform natural language descriptions into STL specifications, aiming to reduce the manual burden and improve accuracy. Existing approaches for transforming natural language requirements into TL and STL specifications can be broadly categorized as rule-based methods and deep learning-based methods. Rule-based methods are widely adopted [23, 32, 48, 51], and they use fixed templates to transform natural language into intermediate representations, which are subsequently transformed into temporal logic formulas through manually designed rules. While these methods reduce human efforts, they not only require extensive expert knowledge and involve a steep learning curve [30], but also are usually limited to highly structured natural language that follows predefined patterns.

With the development of natural language processing, deep learning-based transformation methods have been proposed [6, 19, 25, 31, 45]. The representative approaches include the following: DeepSTL [25] employs a grammar-based data generation technique to train an attentional model that transforms English to STL using transformer-based neural networks. Similarly, NL2TL [6] utilizes LLMs to build the natural language-TL dataset and fine-tunes the T5 model for transformation tasks. KGST [19] uses a two-stage approach: it first fine-tunes an LLM with synthesized diverse data to transform natural language into an initial STL, then refines it using retrieved examples from an external knowledge base. These methods provide impressive improvement in the accuracy of transforming natural language into STL specifications, and also enhance the scalability and generalization.

However, in practical applications, user-written requirements are often vague or ambiguous due to differences of expertise and perspective among users [18, 42, 52]. Following the requirements engineering literature [1, 2, 16, 17, 28, 35, 47], vagueness refers to requirements with underspecified

Vagueness

User: When signal x_1 exceeds 10 between 0 and 15, signal x_2 should decrease until x_1 dropping back.

Candidate STL:

$$\mathbf{F}[0,15](x_1 > 10) \rightarrow \big((x_2 < ?)\ \mathbf{U}\ (x_1 \leq 10)\big)$$

Doubt: "Signal x_2 should decrease" does not specify by how much.

Ambiguity

User: During 10-150 seconds, if signal x_1 exceeds 0.2, then signal x_2 remains less than 0.3 for the next 30 seconds.

Candidate STL 1:

$$\mathbf{F}[10,150](x_1 > 0.2) \rightarrow \mathbf{G}[0,30](x_2 < 0.3)$$

Candidate STL 2:

$$\mathbf{G}[10,150]\big((x_1 > 0.2) \rightarrow \mathbf{G}[0,30](x_2 < 0.3)\big)$$

Doubt: Does "next 30 seconds" start from the first time x_1 exceeds 0.2, or from the whole $[10,150]$ interval?

Fig. 1. User-written natural language requirements may fail to accurately convey their intent. The left part shows vagueness, where the timing constraint is unspecified. The right part illustrates ambiguity, where a single requirement can have multiple interpretations.

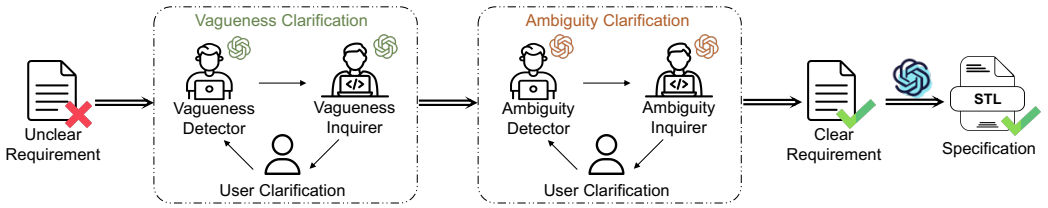

Fig. 2. The Brief Structure of ClarifySTL.

or imprecisely bounded constraints that need further clarification, whereas ambiguity refers to requirements expressed in a way that admits two or more distinct plausible interpretations. As shown in Figure 1, in the case of vagueness, the constraint "signal x_1 should decrease" lacks specific values. In cases of ambiguity, the user input "within the next 30 seconds" can be interpreted in two ways: starting from when x_1 first exceeds 0.2, or within the entire time range from 10 to 150 seconds. Consequently, directly generating STL specifications from such natural language requirements is hard to satisfy the intent of users.

In this paper, to address these challenges, we propose the ClarifySTL framework, which interactively clarifies vagueness and ambiguity in natural language requirements before their transformation into STL. The framework identifies vague and ambiguous components in requirements and generates precise clarifying queries, which can not only avoid unnecessary interactions between LLMs and users, but also guide users in providing effective clarifications. As shown in Figure 2, the ClarifySTL consists of two stages: vagueness clarification and ambiguity clarification. In the vagueness clarification stage, given a natural language requirement, we detect its vagueness by training a Vagueness Detector through fine-tuning LLMs on a multi-classification dataset containing requirements with three types of vagueness: temporal, numerical, and conditional logic, as well as complete requirements. The detector accurately identifies vague expressions, and the Vagueness Inquirer then generates queries to obtain the underspecified information through Chain-of-Thought [53] (CoT)-based prompting. This process iterates until all essential information is provided. Even after the information is clarified, the requirements may still face the risk of multiple possible interpretations, requiring the user to determine which one aligns with their intent. To address this, during the ambiguity clarification stage, the Ambiguity Detector is trained with contrastive learning on LLMs to identify ambiguous expressions, including referential ambiguity and semantic ambiguity. Then, the Ambiguity Inquirer generates multiple STL candidates and

back-translates them into natural language. It compares these natural language versions with the original requirement, analyzes which parts need clarification, and formulates queries for the user. This stage is also iterative until the ambiguity is resolved. Finally, the refined requirements are fed into an LLM to generate the STL specifications.

We evaluate our framework on the representative benchmarks DeepSTL and STL-DivEn [19], as well as on AmbiEval, which is proposed in this work and used to assess the detection and query generation capabilities of the agents. Metric-based evaluation results demonstrate that STL formulas generated from the requirements clarified by our framework achieve higher accuracy than those produced by baselines. For example, ClarifySTL outperforms the current state-of-the-art (SOTA) model by 13.92% and 12.08% in Formula Accuracy and Template Accuracy on the DeepSTL benchmark, and by 12.57% and 12.99% in the same metrics on the STL-DivEn benchmark. Furthermore, our model obtains an average 90.9% detection accuracy for vagueness and ambiguity, which surpasses other baselines. Human evaluation results show that our framework clarifies an average 93.8% of defective requirements. Additionally, 93.3% of the queries generated by our Inquirer are effective, outperforming other methods.

To systematically evaluate the effectiveness of ClarifySTL, we address the following three research questions: ❶ **Transformation Accuracy:** How does ClarifySTL perform compared to existing STL transformation methods in generating accurate STL specifications? ❷ **Detection Capability:** How effectively can ClarifySTL detect vagueness and ambiguity in natural language requirements? ❸ **Query Quality:** How effective and clear are the clarifying queries generated by ClarifySTL for guiding users to provide accurate clarifications?

In summary, our main contributions are as follows:

- We propose ClarifySTL, a novel interactive framework that guides users to clarify vagueness and ambiguity in natural language requirements for STL specifications.
- We introduce vagueness and ambiguity detection modules that accurately identify requirements with the corresponding issues.
- We propose query generation methods that generate targeted queries for users, guiding them to clarify vagueness and ambiguity in requirements.
- We conduct extensive experiments on benchmarks, demonstrating that our approach effectively detects issues, generates clarification queries, and enables users to obtain accurate STL specifications.

Outline. In the rest of this paper, Section 2 provides the background information. Section 3 details our proposed framework, ClarifySTL. Section 4 describes the experimental setup. Section 5 presents the results and their analysis. Section 6 presents further discussion and shows the threats to validity of ClarifySTL. Section 7 reviews related work in this area. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Background: Signal Temporal Logic

STL is a specification logic widely used for CPS. As an illustrative example, consider the *Automatic Transmission (AT)* system, which is frequently employed as a benchmark in system analysis [13–15]. In automotive engineering, AT serves as a key transmission controller that continuously outputs the vehicle's gear, speed, and rpm. One of its safety requirements can be stated as follows: *In the following 12 seconds, whenever the speed is higher than 45 m/s, the engine speed should remain below 2700 rpm within four seconds.* STL is capable of expressing such real-time and real-valued constraints.

Let $\mathbb{R}$ stand for the set of real numbers, $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{+}$ respectively for the nonnegative and positive real numbers. We use $\mathbb{N}_{+}$ to denote the set of positive integers.

Let $T \in \mathbb{R}_+$ be a positive real value, and $d \in \mathbb{N}_+$ be a positive integer. A *d-dimensional signal* is defined as a function $\mathbf{v}\colon [0,T] \to \mathbb{R}^d$, where T represents the *time horizon* of $\mathbf{v}$. Given an arbitrary time instant $t \in [0,T]$, $\mathbf{v}(t)$ is a d-dimensional real vector; each dimension concerns a signal *variable* that has a certain physical property, e.g., speed, rpm, acceleration, etc. In this paper, we select a fixed set X of variables and, without ambiguity, we call a variable a signal (1-dimensional signal).

Definition 1 (STL Syntax). *In STL, both atomic formulas α and general formulas φ are introduced inductively according to the following rules:*

$$\alpha ::\equiv f(x_1,\ldots,x_K) > 0$$
$$\varphi ::\equiv \alpha \mid \bot \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{G}_I\varphi \mid \mathbf{F}_I\varphi \mid \varphi_1\, \mathbf{U}_I\, \varphi_2$$

where f is a K-ary function $f : \mathbb{R}^K \to \mathbb{R}$, $x_1,\ldots,x_K \in X$, and I is a closed non-singular interval in $\mathbb{R}_{\geq 0}$, i.e., $I = [l,u]$, where $l,u \in \mathbb{R}_{\geq 0}$ and $l < u$. $\mathbf{G}$, $\mathbf{F}$, and $\mathbf{U}$ are temporal operators, which are known as always, eventually *and* until, *respectively. The always operator $\mathbf{G}$ and eventually the operator $\mathbf{F}$ can be treated as special instances of the until operator $\mathbf{U}$, which can be defined by $\mathbf{F}_I\varphi \equiv \top\, \mathbf{U}_I\, \varphi$ and $\mathbf{G}_I\varphi \equiv \neg\mathbf{F}_I\neg\varphi$. Other Boolean connectives such as $\vee, \rightarrow$ are provided as syntactic sugar, i.e., $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$.*

The *Boolean semantics* of an STL formula are defined using a satisfaction relation $(\mathbf{v},t) \models \varphi$, indicating that the signal $\mathbf{v}$ satisfies an STL formula φ at the time point t:

$$\begin{aligned}
(\mathbf{v},t) &\models \alpha && \Leftrightarrow && f(\mathbf{v}(t)) \geq 0\\
(\mathbf{v},t) &\models \neg\varphi && \Leftrightarrow && (\mathbf{v},t) \not\models \varphi\\
(\mathbf{v},t) &\models \varphi_1 \wedge \varphi_2 && \Leftrightarrow && (\mathbf{v},t) \models \varphi_1 \wedge (\mathbf{v},t) \models \varphi_2\\
(\mathbf{v},t) &\models \mathbf{G}_{[l,u]}\varphi && \Leftrightarrow && \forall t' \in [t+l, t+u].\, (\mathbf{v},t') \models \varphi\\
(\mathbf{v},t) &\models \mathbf{F}_{[l,u]}\varphi && \Leftrightarrow && \exists t' \in [t+l, t+u].\, (\mathbf{v},t') \models \varphi\\
(\mathbf{v},t) &\models \varphi_1\, \mathbf{U}_{[l,u]}\, \varphi_2 && \Leftrightarrow && \exists t' \in [t+l, t+u].\, (\mathbf{v},t') \models \varphi_2 \wedge \forall t'' \in [t,t'].\, (\mathbf{v},t'') \models \varphi_1
\end{aligned}$$

Now, we are able to formally specify the above *AT safety requirement* using the following STL formula:

$$\mathbf{G}_{[0,12]}(\text{speed} > 45 \rightarrow \mathbf{F}_{[1,4]}(\text{rpm} < 2700)).$$

It should be noted that a *nested* STL formula is an STL formula in which some temporal operator appears within the scope of other temporal operators.

3 Approach

3.1 Overview

The overview of our ClarifySTL model is shown in Figure 3. Our approach consists of two stages: vagueness clarification and ambiguity clarification. Each stage involves a detector and an inquirer. The Vagueness Detector is an LLM fine-tuned on a vagueness detection dataset, and the Vagueness Inquirer generates queries based on CoT prompts; the Ambiguity Detector is built using a contrastive learning-based classifier, and the Ambiguity Inquirer generates clarification queries by analyzing multiple potential interpretations of the input requirement.

After the user inputs a requirement, ClarifySTL first uses the Vagueness Detector to check for vagueness. If vague, it feeds the detected vagueness type and the requirement into the Vagueness Inquirer, which outputs a query to guide the user in clarifying the vagueness. The original requirement, query, and clarification are then sent together to LLMs to generate a supplemented requirement. The refined requirement is rechecked by the Vagueness Detector, and the vagueness

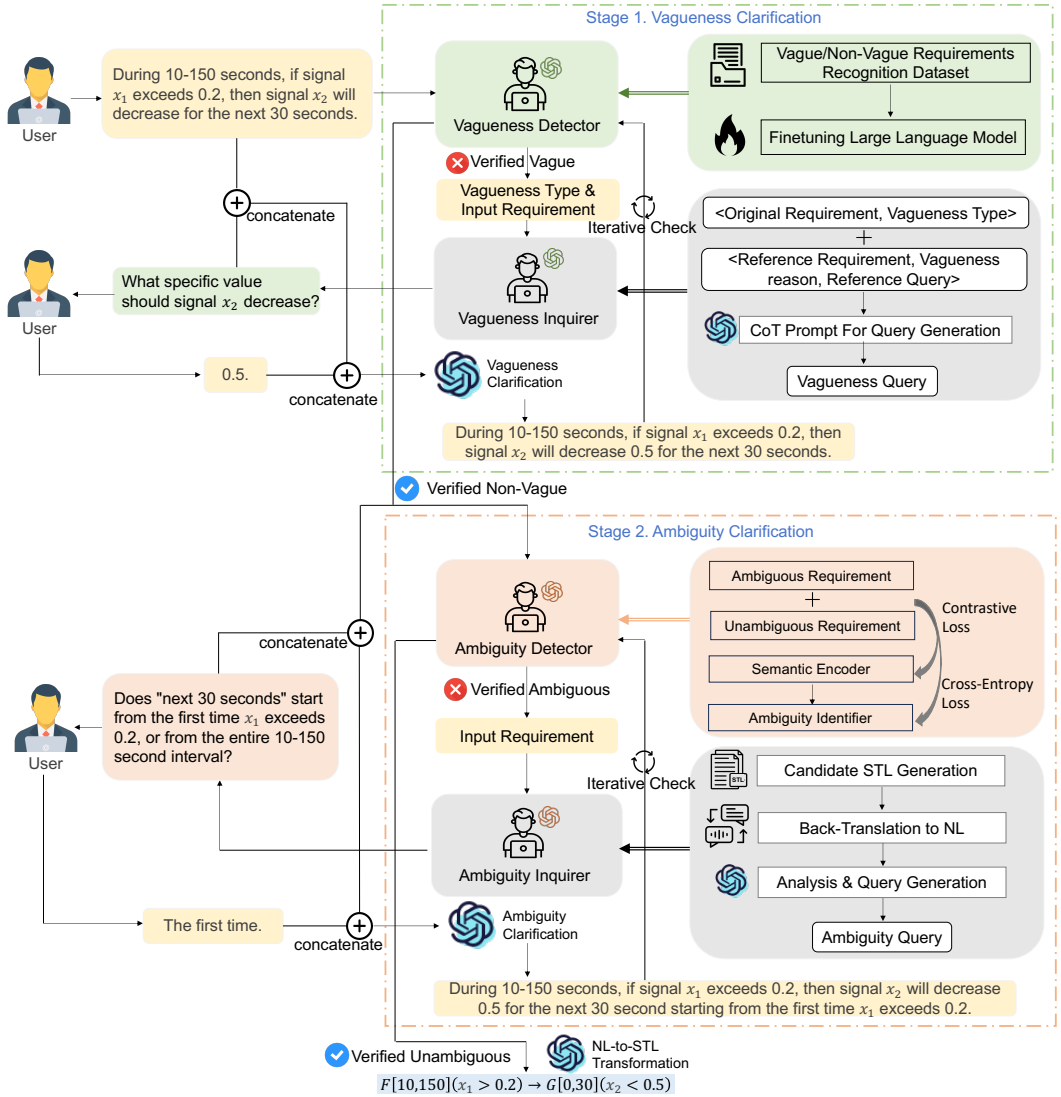

Fig. 3. The Overview of ClaritySTL.

clarification process repeats until no underspecified information is detected. The process then proceeds to the ambiguity clarification stage.

Subsequently, ClarifySTL uses the Ambiguity Detector to check for ambiguity. If ambiguity is detected, the input requirement is sent into the Ambiguity Inquirer, which generates a query to guide the user in clarifying the ambiguity. The original requirement, query, and user's clarification are then sent to LLMs to produce a clarified requirement. Similarly, this process is repeated until no ambiguity is detected in the requirement. Finally, ClarifySTL uses an LLM to transform the clarified requirement into an STL specification. These iterative re-checking loops serve as a post-clarification verification step that allows the user to confirm whether each refinement preserves the original requirement before the process proceeds to the next iteration.

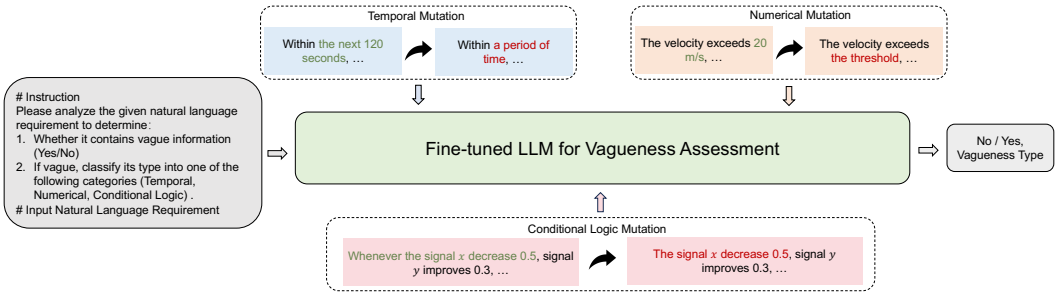

Fig. 4. Fine-tuning the Vagueness Detector for Identification.

Example. As shown in Figure 3, when a user inputs the requirement "During 10–150 seconds, if signal x_1 exceeds 0.2, then signal x_2 will decrease for the next 30 seconds" into the vagueness clarification Stage. The Vagueness Detector identifies that this sentence contains underspecified information and categorizes the type as incomplete numerical information. It then sends this to the Vagueness Inquirer, which outputs the query "What specific value should signal x_2 decrease?". Subsequently, the user responds with 0.5, which is processed along with the original requirement and the query by the LLM to generate a revised requirement: "During 10–150 seconds, if signal x_1 exceeds 0.2, then signal x_2 will decrease 0.5 for the next 30 seconds." After a new round of detection confirms the requirement is complete, it proceeds to the next stage.

After the above vagueness clarification stage, the requirement is first examined by the Ambiguity Detector in the ambiguity clarification stage. The Ambiguity Detector identifies the ambiguity and then sends the requirement to the Ambiguity Inquirer, which generates a query asking about the start time of "the next 30 seconds." After the user clarifies that it means "the first time", the clarification, original requirement, and query are sent to the LLM to produce the clarified requirement: "During 10–150 seconds, if signal x_1 exceeds 0.2, then signal x_2 will decrease 0.5 for the next 30 seconds starting from the first time x_1 exceeds 0.2." The clarified requirement is verified by the Ambiguity Detector to be unambiguous and is finally transformed by the LLM into the STL formula: $\mathbf{F}[10, 150](x_1 > 0.2) \rightarrow \mathbf{G}[0, 30](x_2 < 0.5)$.

3.2 Vagueness Detector

As LLMs lack domain knowledge about vagueness in STL-relevant expressions, they often misjudge precise requirements as vague while overlooking genuinely vague descriptions. To address this issue, we categorize the types of vagueness and fine-tune LLMs on datasets constructed according to these categories, enabling LLMs to accurately identify vague information. Since STL syntax in Definition 1 is built from temporal intervals, numerical predicates, and logical compositions, we define vagueness by the underspecified information needed to instantiate these STL components. Accordingly, for NL-to-STL transformation, we categorize vagueness into the following three types:

(1) **Temporal Information Vagueness.** Temporal constraints in natural language requirements, such as duration, start time, or time intervals, are often partially omitted or not expressed with explicit temporal information, but instead are conveyed using terms like "soon", "later", or "in a moment" [41]. This lack of precision makes it impossible to accurately construct precise temporal constraints in the corresponding STL formulas.
(2) **Numerical Information Vagueness.** Expressions involving numerical values (e.g., rpm, speed, temperature) are often omitted or replaced by qualitative descriptors in natural language

requirements [41], resulting in missing numeric thresholds in the corresponding STL formulas and making it impossible to fully define the quantitative aspects of system behavior.

(3) **Conditional Logic Information Vagueness.** Conditional logic in natural language is often partially omitted or insufficiently expressed. This vagueness reduces the precision of conditional judgments and makes it difficult to infer the correct logical operators during transformation into STL formulas, affecting the accuracy and effectiveness of the resulting formal specification.

Subsequently, we construct the AmbiEval dataset, which consists of two parts. Here, we introduce Part A of the dataset, which contains vague requirements and their types of incompleteness, and the corresponding complete requirements. For each selected input from DeepSTL and STL-DivEn, we first ask the teacher model to identify STL-relevant elements that can be perturbed according to our vagueness taxonomy. If multiple valid perturbation opportunities exist in the same requirement, the teacher model processes them iteratively and introduce multiple vague elements into the same mutated requirement. Each mutated requirement therefore contain more than one type of vague information. All valid mutated requirements are retained in AmbiEval.

Specifically, we employ GPT-4o as the teacher model and guide it with carefully designed mutation prompts to rewrite the original requirements in a directed, type-specific manner. To ensure linguistic validity, we require that the teacher model "keep grammatical correctness and semantic coherence" in prompts, and we further implement a rule-based syntactic validator that detects grammatical issues and automatically filters out mutants that fail the check. Let V_{T}, V_{N}, and V_{C} denote the expression sets used for temporal, numerical, and conditional-logic vagueness, respectively. For each mutation, w denotes a vague or underspecified expression sampled from the corresponding set.

- **Temporal Mutation.** Remove or replace the time interval in the requirement with a vague description. Formally, when the corresponding STL formula contains a bounded temporal operator $G_{[a,b]}$, $F_{[a,b]}$, or $U_{[a,b]}$, the interval $[a, b]$ in the aligned span of the requirement is either deleted or replaced by a vague phrase $w \in V_{\mathrm{T}}$ (e.g., replace an explicit "[0, 10] time units" with "within the next period of time").
- **Numerical Mutation.** Replace numeric thresholds or concrete values in the requirement with vague descriptions, or delete them directly. Formally, when the corresponding STL formula contains an atomic predicate $f(x_1, \ldots, x_K) \sim c$ with $c \in \mathbb{R}$, the threshold c in the aligned span is either deleted or replaced by a qualitative descriptor $w \in V_{\mathrm{N}}$ (e.g., change "speed exceeds 50" to "speed is high").
- **Conditional Logic Mutation.** Delete the logical or conditional relationship in the requirement or replace it with an underspecified expression. Formally, when the corresponding STL formula contains a logical connective $\circ \in \{\neg, \wedge, \vee, \rightarrow\}$, the expression that corresponds to $\circ$ in the requirement is either deleted or replaced by an underspecified expression $w \in V_{\mathrm{C}}$ (e.g., change "if speed > 50 then brake activates" to "speed > 50, brake activates").

We then fine-tune an LLM on this dataset to obtain an Vagueness Detector capable of identifying which requirements contain underspecified information, as shown in Figure 4. Specifically, each training pair in the dataset is structured with three attributes: `Instruction`, `Input`, and `Output`. The `instruction` provides the guidance to determine whether a natural language requirement is incomplete and to classify the type of vagueness. The `input` is the natural language requirement to be evaluated and is provided to the LLM together with the instruction. The LLM is trained to generate the judgment result of the input as the `output`. After training, the model functions as the Vagueness Detector.

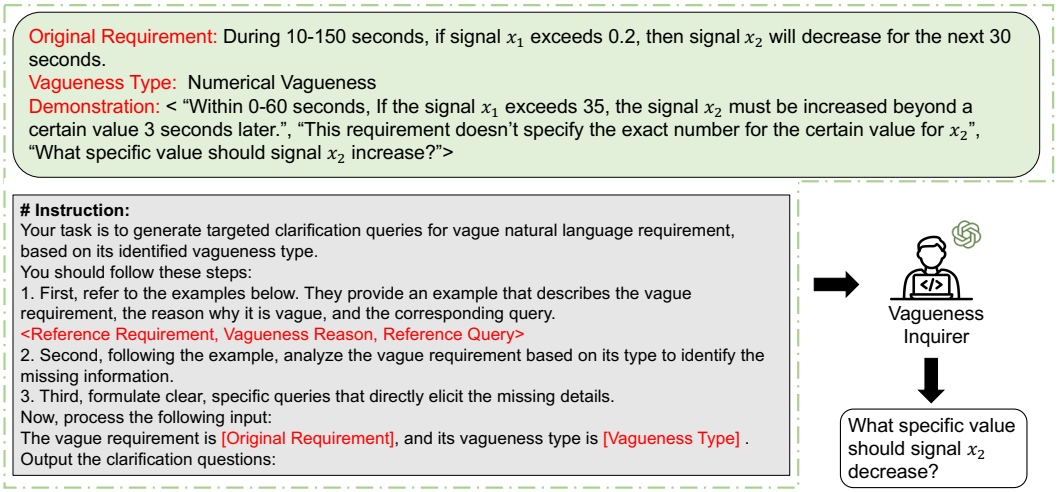

Fig. 5. Prompts for the Vagueness Inquirer to Generate Clarification Queries.

3.3 Vagueness Inquirer

Precise clarification questions help users clearly express their intentions and ensure the information targets the incomplete parts of the requirements. Vague or overly broad questions may confuse users, produce irrelevant answers, and increase their burden, making it difficult to provide effective clarifications or causing them to give incorrect ones. [42]. Therefore, once vague requirements are identified, it is crucial to equip LLMs with the ability to generate precise questions.

To achieve this, we design a Vagueness Inquerier to automatically generate targeted clarification questions for vagueness clarification. The agent leverages a CoT prompting strategy, enabling step-by-step reasoning over vague requirements. As shown in Figure 5, the prompt consists of three components: Instruction, Requirement Input, and Demonstrations.

(1) **Instruction.** Specifies the task for the model, i.e., generating clarification questions based on vague requirements and their error types.
(2) **Requirement Input.** Provides the vague user requirements along with their vagueness types to the model.
(3) **Demonstrations.** Selects examples corresponding to the requirement's error type, enabling the model to learn in context from <Reference Requirement, Incompleteness Reason, Reference Query> triples and to formulate targeted questions that elicit the underspecified information needed to construct complete STL formulas.

By structuring the agent in this way, it can effectively leverage the reasoning and text generation capabilities of LLMs to address multiple forms of natural language vagueness, ensuring that the generated questions are both specific and informative.

To prevent the Vagueness Inquirer from altering the user's intent, we explicitly add a constraint in its prompt that requires the generated clarification questions to focus only on the vagueness type and the missing components reported by the Vagueness Detector, and not to introduce information unrelated to the detected vague segments of the requirement. In addition, the prompt allows the user to indicate that the requirement does not contain vagueness if a non-vague sentence is mistakenly fed to the Vagueness Inquirer. In such cases, the agent directly returns the original requirement without adding any supplementary information and passes it to the next stage. When the user's

answer fails to effectively supplement the targeted information, the system asks the user to restate the intent.

3.4 Ambiguity Detector

Ambiguous and unambiguous natural language may exhibit high similarity in syntactic structure and lexical expressions, such as similar sentence patterns, frequently used terms, or descriptive styles, which makes distinguishing them challenging. The main difficulty lies in capturing their representative features from subtle differences in word usage and semantics. Therefore, we propose to encode both ambiguous and unambiguous requirements using contrastive learning and subsequently classify them based on their encoded representations.

Hoffer and Ailon [26] introduce the triplet network as a method for contrastive representation learning. The network input is a triplet (x, x^+, x^-), where x is a given natural language sentence (i.e., the anchor), x^+ is a positive sample from the same class, and x^- is a negative sample from a different class. The triplet network learns useful representations for downstream classification tasks by minimizing the distance between x and x^+ while maximizing the distance between x and x^-. In order to more effectively encode representations of ambiguous and unambiguous requirements, we adopt a triplet network-based contrastive learning approach to build the Ambiguity Detector.

Ambiguity in NL-to-STL transformation mainly affects either the referenced entity or the interpretation of temporal, logical, and relational constraints; accordingly, we categorize ambiguity into two types: referential ambiguity and semantic ambiguity. We use unambiguous requirements from the DeepSTL and STL-DivEn datasets and, following the same mutation pipeline as in Section 3.2, introduce controlled ambiguities by prompting GPT-4o under two type-specific mutation operators to generate their ambiguous variants:

- **Referential Ambiguity Mutation.** Replace an explicit signal name in the requirement with a pronoun or a generic noun phrase, so that the signal referent becomes underdetermined. Formally, when the corresponding STL formula contains $K \geq 2$ distinct signal identifiers $\{x_1, \ldots, x_K\}$, an occurrence of a signal identifier in the aligned span is replaced by a referring expression $w \in V_R$, where V_R denotes the set of pronouns and generic noun phrases used for referential ambiguity and w is a sampled expression from this set (e.g., change "x_1 exceeds 0.2 and x_1 stays high" to "x_1 exceeds 0.2 and it stays high").
- **Semantic Ambiguity Mutation.** Rewrite the requirement so that the involved signals and predicates remain unchanged, but the sentence admits multiple plausible STL interpretations. This mutation is realized through in-context learning: we prompt GPT-4o with a fixed set of demonstration triples *(original requirement, ambiguous rewrite, ambiguity explanation)* and ask it to produce a similar rewrite for each new input requirement. For example, "x_2 should stay below 0.5 within 30 seconds after x_1 exceeds 0.2" can be rewritten as "within the next 30 seconds, if x_1 exceeds 0.2, then x_2 should stay below 0.5", where "the next 30 seconds" admits both "starting from when x_1 exceeds 0.2" and "starting from the current moment". The full prompts are provided in the supplementary file.

The generated mutant is first filtered by syntactic validator and then manually validated by two annotators with at least three years of experience in writing STL specifications, who independently verify that the mutant indeed admits multiple plausible STL interpretations. This forms Part B of the AmbiEval, which includes the ambiguous natural language requirements, their corresponding unambiguous requirements, and labels, where ambiguous requirements are labeled as 1 and unambiguous ones as 0.

As shown in Figure 6, for a given sentence x, we assume that x^+ belongs to the same class as x, while x^- belongs to a different class. Specifically, if x is an ambiguous requirement, x^+ is also

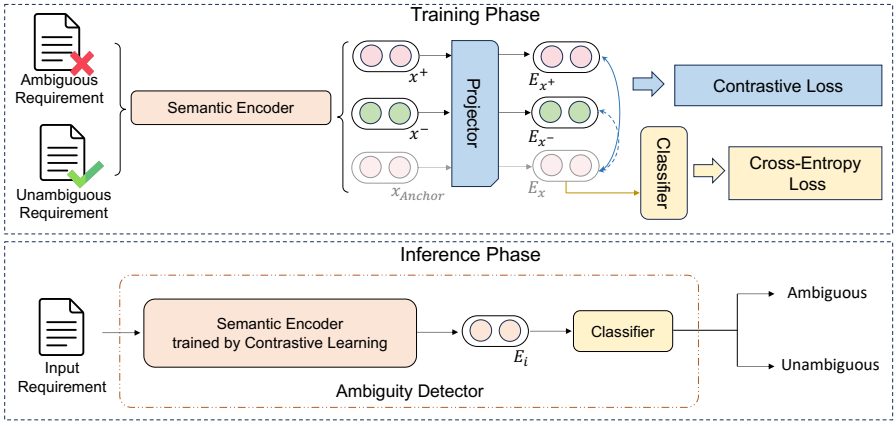

Fig. 6. Training and Inference Phases of the Ambiguity Detector.

ambiguous, whereas x^- is unambiguous. Our model uses the hidden states of LLaMA 3-8B as the base model to obtain embeddings for x, x^+, and x^-. To perform contrastive learning, we first follow prior works [22, 55] and forward the same input twice with different hidden-layer dropout masks (dropout probability = 0.1) to generate x and x^+.

Specifically, the original sentence x is encoded twice with dropout using LLaMA 3-8B as the semantic encoder. We extract the hidden states from its last transformer layer and apply mean pooling over all token representations to obtain two distinct sentence-level representations, which serve as the anchor x and the positive instance x^+ for contrastive learning [26]. During this process, the parameters of the semantic encoder are frozen in order to preserve the pretrained semantic knowledge, and only the projector and the subsequent lightweight classification head are updated. Subsequently, we use a structure called a *projector* to transform the outputs of the semantic encoder into final embeddings E_x, E_{x^+}, and E_{x^-}. A contrastive loss is then computed based on the cosine distance of the embeddings to train the network. The training objective is to minimize the distance between x and x^+ while separating x and x^-, which is formulated as:

$$\mathcal{L}_c = \max\left(\|E_x - E_{x^+}\| - \|E_x - E_{x^-}\| + \epsilon, 0\right)$$

where ϵ denotes the margin between x and x^-, with a default value of 1.

By using contrastive learning, the Ambiguity Detector is able to learn meaningful representations E_x for an input natural language requirement x. These embeddings are then passed through a classification layer with the softmax function to obtain the probability for each class. A threshold of 0.5 is applied to decide whether the sentence is ambiguous or unambiguous.

3.5 Ambiguity Inquirer

When the natural language requirements are identified as ambiguous, we generate targeted queries to guide the user to provide clarifications. To avoid asking the user to rewrite the entire requirement, we design a workflow that leverages LLMs to locate ambiguous parts and generate corresponding clarification queries, as shown in Fig. 7. First, we leverage LLMs to perform multiple samplings of the input original requirement to obtain STL specification candidates. These candidates are back-translated into natural language, and LLMs analyze the differences between them and the

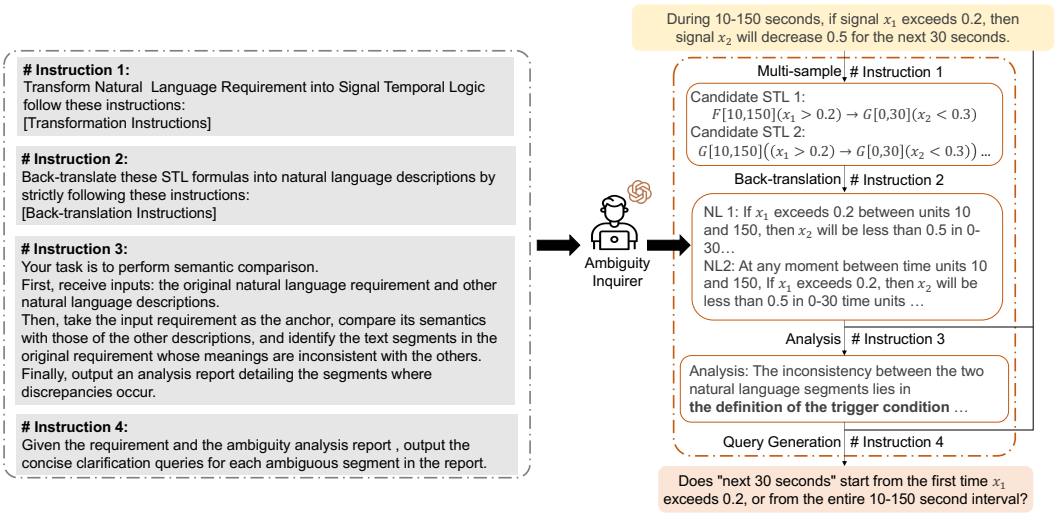

Fig. 7. Workflow and Prompts for Ambiguity Inquirer to Generate queries about Ambiguity Clarification.

original requirement to generate a report. Finally, based on this report, the LLM formulates clarification queries. The prompts for STL transformation and back-translation can be found in the supplementary file. The details are as follows:

- **Candidate STL Generation.** To better help the LLMs locate ambiguous portions, we first sample the same original requirement multiple times under the same prompt and parameters, generating a set of candidate STL formulas $\{\phi_1, \phi_2, \ldots, \phi_N\}$, representing different possible interpretations of the same requirement. By comparing the differences among these specifications, potential points of ambiguity in the requirement can be preliminarily located.
- **Back-Translation to Natural Language.** Since LLMs have limited capability for direct symbolic-level analysis [7], we back-translate the STL candidates into the corresponding natural language descriptions $\{d_1, d_2, \ldots, d_N\}$ and leverage the strong reasoning capabilities of LLMs in natural language to conduct analysis. To minimize interference from lexical variability, each prompt enforces a fixed translation scheme for variables and symbols, ensuring that the generated descriptions are uniform and facilitating semantic comparison by LLMs.
- **GPT-Based Difference Analysis.** The original requirement R and all candidate descriptions $\{d_1, d_2, \ldots, d_N\}$ are fed into an LLM, which performs a comprehensive semantic comparison across all input natural language descriptions and generates a discrepancy analysis report. The report will reveal the differences within these natural language descriptions, enabling the model to locate which parts of the original requirement R may introduce ambiguity.
- **Clarification Query Generation.** Finally, we provide the original natural language requirement and the discrepancy analysis report to the LLMs, prompting them to generate precise queries for the user to clarify the ambiguity in the requirement.

We further add a constraint in the prompt that restricts the generated queries to the divergence points identified in the discrepancy analysis report, so that the queries do not introduce information unrelated to the detected ambiguous segments. In addition, the prompt allows the user to indicate that the requirement does not contain ambiguity if an unambiguous sentence is mistakenly provided to the Ambiguity Inquirer. In such cases, the agent directly returns the original requirement without

modifying any content and passes it to the next stage. When the user's answer fails to effectively clarify the ambiguous part, the system asks the user to restate the intent.

4 Study Design

To evaluate the effectiveness of our ClarifySTL , we design three research questions (RQ) and perform studies to answer these questions. In this section, we describe the research questions and details of our study. We present the evaluation results and analysis in Section 5.

4.1 Research Questions

We address the following three research questions to systematically evaluate the performance and effectiveness of ClarifySTL.

RQ1: How does ClarifySTL perform compared to existing STL transformation methods? This research question aims to evaluate whether ClarifySTL generates more accurate and semantically consistent STL specifications than existing methods. To achieve this, we compare ClarifySTL against two types of methods: 11 baselines that directly transform natural language into STL, and five LLM–based interactive transformation methods. The comparison is conducted on two representative STL transformation benchmarks, including DeepSTL and STL-DivEn.

RQ2: How effective is ClarifySTL at detecting vagueness and ambiguity in natural language requirements? The ability to detect vagueness and ambiguity is critical for ensuring the reliability of ClarifySTL framework. In this research question, we investigate ClarifySTL's capability to identify vague and ambiguous requirements.

RQ3: Are the queries generated by ClarifySTL effective and clear for users to provide clarification? Generating high-quality clarification queries can significantly reduce the user's effort. In this research question, we evaluate whether the queries generated by our framework accurately highlight the vague or ambiguous portions of the requirement and whether they are easy for users to understand.

4.2 Datasets

Our datasets include two representative Natural Language-to-Signal Temporal Logic (NL-to-STL) datasets and one dataset designed for detection and query generation tasks. Specifically, we adopt two NL-STL datasets, DeepSTL [25] and STL-DivEn [19], to evaluate the model's ability to perform NL-to-STL transformation. Additionally, in this work, we propose the AmbiEval dataset. It is used to fine-tune models for detecting vagueness and ambiguity, as well as to evaluate ClarifySTL's performance in identification and query generation.

- **DeepSTL** is a synthetic dataset generated by rule-based methods. To ensure systematic coverage, DeepSTL defines a structured fragment of STL formulas with three layers and performs random sampling according to empirical operator distributions. Each sampled STL formula is then translated into multiple natural language requirements using a template-based translation procedure, enriched with synonymous expressions to capture linguistic variability.
- **STL-DivEn** is a dataset designed to enhance the diversity and quality of NL-STL pairs. STL-DivEn begins with a handcrafted seed set of 120 NL-STL pairs covering both basic and nested logic. Representative samples are selected through clustering and used to guide LLMs in generating new pairs. The generated data are refined using rule-based filters and human validation, and the validated pairs are iteratively added to expand the dataset.
- **AmbiEval** is a dataset adapted from DeepSTL and STL-DivEn, used to fine-tune LLMs for detecting vagueness and ambiguity and to evaluate the performance of models in detection and query generation. It contains 9,200 labeled instances: Part A includes 6,000 instances for

Table 1. Number of Requirements with Vagueness and Ambiguity Identified in the DeepSTL and STL-DivEn Datasets. "Both" Denotes Requirements Containing Both Vagueness and Ambiguity.

Dataset	Vagueness			Ambiguity		Both
	Temporal	Numerical	Conditional Logic	Referential	Semantic	
DeepSTL	87	39	103	83	117	62
STL-DivEn	67	56	43	71	212	77

vagueness detection across three types of vagueness, and Part B includes 3,200 instances for ambiguity detection, covering 1,600 instances of referential ambiguity and 1,600 instances of semantic ambiguity. Each vague instance has a corresponding non-vague version, and each ambiguous instance has a corresponding unambiguous version. Each instance is also paired with a human-written query, serving as the ground truth for query generation.

We randomly select 14,000 samples from DeepSTL and STL-DivEn as the training set and 2,000 samples as the test set for NL-to-STL evaluation. To support the human evaluation in RQ1, we further identify requirements containing vagueness or ambiguity from the NL-to-STL evaluation data. Table 1 summarizes the distribution of these selected defective requirements. In DeepSTL, the selected subset contains 87 temporal, 39 numerical, and 103 conditional-logic vagueness instances, as well as 83 referential and 117 semantic ambiguity instances, with 62 requirements containing both vagueness and ambiguity. In STL-DivEn, the selected subset contains 67 temporal, 56 numerical, and 43 conditional-logic vagueness instances, together with 71 referential and 212 semantic ambiguity instances, with 77 requirements containing both types of issues.

AmbiEval is constructed separately for training and evaluating the vagueness and ambiguity detectors, as well as the query generation modules. To prevent data leakage, AmbiEval is constructed exclusively from the training split of DeepSTL and STL-DivEn and has no overlap with the 2,000 test samples used for NL-to-STL evaluation. For AmbiEval, we perform a random selection with an even distribution across each category in Part A, resulting in 5,400 samples for training and 600 for testing. Similarly, in Part B, 2,800 samples are selected for training and 400 for testing.

4.3 Evaluation Metrics

Following previous works [19, 25], we adopt formula accuracy, template accuracy, and BLEU [43] as evaluation metrics to assess the performance of NL-to-STL. Before computing these metrics, all generated STL formulas are first checked by a standard STL syntax checker, and formulas that fail to pass the STL syntax checker are directly considered incorrect.

Formula Accuracy measures token-level alignment between the generated STL formula and the reference formula. Specifically, we split both formulas into STL tokens, compare the tokens at the same positions, and compute the proportion of reference tokens correctly matched by the generated formula. Template Accuracy first extracts both formulas into STL templates by replacing concrete signal names, predicates, numerical thresholds, and time bounds with template symbols, and then computes the same token-level alignment score over the resulting templates. BLEU is originally designed for machine translation and evaluates n-gram overlap, reflecting lexical semantic similarity.

In addition to string-level metrics, STL semantic robustness provides a complementary way to check the semantic consistency of the generated formulas. In our setting, for each ground-truth STL formula, we first generate a set of signal traces according to its variables, temporal horizons, and numerical thresholds. These traces are sampled to include satisfying and violating cases with respect to the ground-truth formula. The semantic robustness score is then computed as the percentage of

traces on which the generated formula and the ground-truth formula are either both satisfied or both violated.

For vagueness and ambiguity detection tasks, i.e., classification tasks, we use Accuracy, Precision, Recall, and F1-score. Accuracy measures the proportion of correctly predicted instances. Precision measures the proportion of correctly predicted positive instances among all instances predicted as positive, while Recall measures the proportion of actual positive instances that are correctly identified. F1-score is the harmonic mean of precision and recall, providing a balanced measure of model performance.

For query generation tasks, we use ROUGE [33] and BERTScore [59]. ROUGE evaluates the overlap between generated text and reference text, commonly used in text generation tasks. BERTScore computes the semantic similarity between generated text and reference text using contextual embeddings from pre-trained language models.

4.4 Baselines

We compare ClarifySTL with 16 baseline methods, which are grouped into direct transformation and interactive transformation.

Direct Transformation (Open-source Models).

- **DeepSTL** is the pioneering work that first applied deep learning to learn STL transformations from NL-to-STL datasets.
- **KGST** is the current SOTA model, featuring a two-stage framework: it fine-tunes an LLM on NL-to-STL data to generate preliminary STL formulas, then refines them using retrieved reference examples through LLMs.
- **Qwen 3-8B** and **Qwen 3-14B** [58] are LLMs based on a Mixture-of-Experts architecture with optimized inference efficiency, demonstrating strong instruction-following capabilities.
- **LLaMA 3-8B** and **LLaMA 3-14B** [10] are general-purpose models trained on large-scale data, excelling in text generation and reasoning.

Direct Transformation (Closed-source Models).

- **DeepSeek-V3** [34] is an advanced Mixture-of-Experts model employing multi-stage training to achieve strong reasoning and text generation performance.
- **GPT-4o** [27] is one of the most powerful LLMs, achieving SOTA results in formal reasoning tasks.
- **GPT-4o-mini** is a compact version of GPT-4o, optimized for speed and efficiency while retaining most reasoning capabilities.
- **Gemini-2.5-Pro** [8] is a large language model excelling in reasoning and text generation tasks.
- **Claude-4-Sonnet** [1] is a high-performance version of the Claude-4 series, performing reliably on formal reasoning tasks.

Among the direct transformation baselines, DeepSTL and KGST follow their original implementations. For the remaining LLM-based baselines (both open-source and closed-source), we adopt few-shot prompts that include representative NL-STL pairs as in-context examples, along with explicit task instructions and the STL grammar definition, to guide the models in performing the transformation. To ensure fair comparison, we set the temperature to 0 for deterministic outputs. All prompt templates are provided in the supplementary file.

Interactive Transformation. Interactive transformation follows the same two-stage workflow (vagueness clarification followed by ambiguity clarification) as ClarifySTL, but replaces ClarifySTL's

[1] https://www.anthropic.com/news/claude-4

specialized modules with general-purpose LLM capabilities. Specifically, ClarifySTL employs a fine-tuned Vagueness Detector, a contrastive learning-based Ambiguity Detector, and task-specific query generation strategies (CoT-based prompting for vagueness and back-translation-based analysis for ambiguity). In contrast, the interactive transformation baselines use a single unified few-shot prompt to instruct LLMs to perform all tasks, including detecting vagueness and ambiguity and generating clarification queries, relying solely on the intrinsic knowledge of LLMs without any fine-tuned detectors or specialized strategies. In our preliminary experiments, we find that open-source models yield poor performance under this setting; therefore, we only use the aforementioned five closed-source models as interactive transformation baselines.

To simulate a realistic feedback process, we recruit 10 participants, all of whom are graduate students or researchers in the field of CPS with at least three years of experience in writing and interpreting STL specifications. Before the evaluation, each participant receives a guideline document that describes the task objective, the format of the clarification queries, and examples of expected responses. Each participant is provided with a set of requirements and automatically generated clarification queries, and is asked to answer these queries. Each defective requirement is clarified by three participants, and the experimental results are reported as the average of all responses. To assess the consistency among participants, we evaluate agreement from two perspectives. First, for the correctness judgments of the final STL formulas, we compute Fleiss' Kappa [20], obtaining a score of 0.84, indicating agreement. Second, for the clarification responses, we measure the pairwise match rate of the STL formulas generated from different participants' clarifications of the same requirement. The average pairwise match rate is 87.2%, confirming that participants provide consistent clarifications.

4.5 Implementation Details

Our experiments are conducted on 8 NVIDIA A100 GPUs with 40GB of memory. We implement our framework using PyTorch [44], Huggingface Transformers [54], and LLaMA-Factory [60]. In ClarifySTL, GPT-4o is used for vagueness query generation, ambiguity query generation, and requirement refinement. Our framework uses four LLaMA 3-8B models. The Vagueness Detector and the two NL-to-STL transformation models are fine-tuned on the corresponding datasets, while the Ambiguity Detector uses a frozen LLaMA 3-8B encoder together with a trainable projector and a lightweight classification head. The trainable models are optimized for 10 epochs using Adam [29], with a learning rate of 5e-5 and a batch size of 16.

For the Ambiguity Detector, the projector is implemented as a two-layer MLP with ReLU activation, whose dimensions are 4096→1024→256, followed by L2 normalization. For detection tasks, we use few-shot prompts that include the task definition (i.e., identifying whether a requirement is vague or ambiguous), the category descriptions, and representative labeled examples for each category. We evaluate 1, 3, 5, and 8 in-context examples and use 3 as the default for its best overall performance–token efficiency. For clarifying query generation, the maximum number of generated tokens is set to 300. The temperature is set to 0 by default, except when sampling candidate STL formulas in ambiguity clarification, where it is set to 0.9.

For KGST, we fine-tune LLaMA 3-8B separately on DeepSTL and STL-DivEn, and use GPT-4o to refine the preliminary STL formulas. The training hyperparameters are the same as those used in ClarifySTL. For the interactive transformation baselines, we use a unified few-shot prompt to guide LLMs in defect detection and query generation. The prompts for transformation, requirement refinement, and interactive transformation are provided in the supplementary file.

5 Results and Analyses

In this section, we answer the three research questions based on our experimental results.

Table 2. Metric-based Evaluation of NL-to-STL Transformation on DeepSTL and STL-DivEn. Models Are Grouped into Three Categories: Direct Transformation with Open-source Models, Direct Transformation with Closed-source Models, and Interactive Transformation. Acc. Stands for Accuracy, and Sem. Rob. denotes semantic robustness. Bold Values Indicate the Best Results; Red Values Show ClarifySTL's Improvements over the Previous SOTA.

Model	DeepSTL				STL-DivEn			
	Formula Acc. (%)	Template Acc. (%)	BLEU	Sem. Rob. (%)	Formula Acc. (%)	Template Acc. (%)	BLEU	Sem. Rob. (%)
Direct Transformation (Open-Source Models)								
DeepSTL	20.02	29.16	0.3332	35.6	21.35	22.97	0.0312	34.7
KGST	52.81	57.13	0.5272	70.3	57.24	58.83	0.2283	73.6
Qwen 3-8B	28.83	30.81	0.4121	44.8	41.74	43.28	0.0192	57.6
Qwen 3-14B	37.28	39.21	0.4982	54.2	50.82	51.12	0.2092	66.4
LLaMA 3-8B	30.21	32.84	0.4394	46.7	44.92	45.73	0.0231	60.5
LLaMA 3-14B	39.92	41.82	0.4762	56.4	53.82	54.92	0.2183	69.0
Direct Transformation (Closed-source Models)								
DeepSeek-V3	32.97	34.82	0.4182	49.1	51.93	53.72	0.1093	66.8
GPT-4o	30.82	32.77	0.3921	47.8	55.24	57.93	0.1982	70.2
GPT-4o-mini	28.72	29.82	0.3728	43.5	52.83	53.82	0.1872	67.1
Gemini-2.5-Pro	35.81	38.21	0.4682	53.1	56.82	58.32	0.1877	72.4
Claude-4-Sonnet	40.12	40.22	0.5135	58.4	55.43	57.93	0.1903	71.3
Interactive Transformation								
DeepSeek-V3	38.74	39.82	0.4316	55.7	50.26	50.97	0.1221	63.2
GPT-4o	34.86	37.64	0.4211	52.3	53.71	53.86	0.2046	67.5
GPT-4o-mini	32.57	34.38	0.3859	48.6	49.76	51.68	0.1957	63.5
Gemini-2.5-Pro	40.62	42.31	0.4826	58.1	53.67	55.41	0.2117	68.9
Claude-4-Sonnet	45.68	47.41	0.5162	63.2	54.67	55.89	0.2014	69.5
ClarifySTL	**67.12** (14.31 ↑)	**70.03** (12.90 ↑)	**0.5736** (0.0464 ↑)	**82.7** (12.4 ↑)	**70.74** (13.50 ↑)	**72.28** (13.45 ↑)	**0.2612** (0.0329 ↑)	**84.9** (11.3 ↑)

Table 3. Human Evaluation of Interactive Transformation Methods: STL Transformation Accuracy (%) on Requirements with Different Types of Vagueness and Ambiguity, Evaluated on DeepSTL and STL-DivEn.

Dataset	Model	Vagueness Type			Ambiguity Type	
		#Temporal Acc. (%)	# Numerical Acc. (%)	# Condition Acc. (%)	#Referential Acc. (%)	#Semantic Acc. (%)
DeepSTL	DeepSeek-V3	73.8	75.7	74.2	78.4	71.3
	GPT-4o	74.1	77.2	75.5	80.1	72.8
	GPT-4o-mini	65.9	66.4	68.2	67.8	62.9
	Gemini-2.5-Pro	73.7	76.4	77.3	80.7	73.1
	Claude-4-Sonnet	74.8	76.3	78.4	82.4	74.3
	ClarifySTL	**91.5**	**95.2**	**93.7**	**96.1**	**92.8**
STL-DivEn	DeepSeek-V3	66.8	68.3	67.3	72.4	59.2
	GPT-4o	67.3	69.7	66.3	73.1	60.6
	GPT-4o-mini	61.8	62.1	61.5	68.2	58.3
	Gemini-2.5-Pro	69.2	69.3	65.5	74.3	61.4
	Claude-4-Sonnet	69.8	68.2	66.9	75.7	61.1
	ClarifySTL	**92.8**	**96.7**	**94.2**	**98.4**	**93.4**

5.1 RQ1: How does ClarifySTL perform compared to existing STL transformation methods?

To evaluate the effectiveness of ClarifySTL, we design experiments to compare its performance with 16 baselines in the NL-to-STL task, using DeepSTL and STL-DivEn as benchmarks. The experimental setup is detailed in Sections 4.4 and 4.5.

5.1.1 Metric-Based Evaluation. Table 2 presents the metric-based evaluation results comparing ClarifySTL with other baselines. Bold numbers denote the best performance, and red values represent ClarifySTL's relative improvements over SOTA.

We observe that ClarifySTL achieves the best performance across all evaluation datasets. Compared with closed-source LLMs in the Direct Transformation category, it shows performance gains on both datasets. On DeepSTL, ClarifySTL surpasses the best-performing model, Claude-4-Sonnet, by 27.00% in Formula Accuracy, 29.81% in Template Accuracy, and 0.0601 in BLEU. On STL-DivEn, it outperforms Gemini-2.5-Pro by 13.92% in Formula Accuracy and 13.96% in Template Accuracy, and exceeds GPT-4o by 0.0630 in BLEU. Compared to models in the Interactive Transformation

category, ClarifySTL also shows improvements. On DeepSTL, it surpasses Claude-4-Sonnet by 21.44% in Formula Accuracy, 22.62% in Template Accuracy, and 0.0574 in BLEU. On STL-DivEn, ClarifySTL outperforms Claude-4-Sonnet by 16.07% in Formula Accuracy and 16.39% in Template Accuracy, and surpasses Gemini-2.5-Pro by 0.0495 in BLEU. Compared with the SOTA model KGST, ClarifySTL also achieves improvements. On DeepSTL, it surpasses KGST by 14.31% in Formula Accuracy, 12.90% in Template Accuracy, and 0.0464 in BLEU. On STL-DivEn, the improvements reach 13.50%, 13.45%, and 0.0329, respectively. For semantic robustness, ClarifySTL also obtains the highest scores on both datasets, reaching 82.7% on DeepSTL and 84.9% on STL-DivEn. Compared with KGST, it improves semantic robustness by 12.4% and 11.3%, respectively. This indicates that the formulas generated by ClarifySTL not only match the reference formulas more accurately at the syntactic level, but also better preserve the expected STL satisfaction behavior on sampled signal traces.

These results demonstrate that: (1) ClarifySTL uses requirement clarification to handle vagueness and ambiguity, achieving better performance than other methods. It outperforms direct transformation methods that transform natural language to STL without clarification, as well as interactive transformation approaches that rely solely on the intrinsic knowledge of LLMs. (2) Interactive methods perform better than direct transformation. This indicates that optimizing the input natural language enables the models to generate more accurate STL formulas. It also suggests that in some cases, the lower accuracy of generated STL is partly caused by vague or ambiguous input, which makes it difficult for LLMs to perform correct transformations.

5.1.2 Human Evaluation. Table 3 presents the human evaluation results of different interactive methods in handling various types of vagueness and ambiguity. Taking the DeepSTL benchmark as an example, ClarifySTL achieves the highest accuracy among all methods across temporal, numerical, and conditional-logic vagueness, with accuracies of 91.5%, 95.2%, and 93.7%, respectively. It also achieves 96.1% accuracy on referential ambiguity and 92.8% on semantic ambiguity. These results demonstrate that our framework is highly effective in handling both vagueness and ambiguity in requirements.

Answer to RQ1. ClarifySTL outperforms existing STL transformation methods, including the SOTA model KGST. On the DeepSTL dataset, it exceeds KGST by **14.31%** in Formula Accuracy, **12.90%** in Template Accuracy, **0.0464** in BLEU, and **12.4%** in semantic robustness. On the STL-DivEn dataset, it outperforms KGST by **13.50%**, **13.45%**, **0.0329**, and **11.3%**, respectively. The results of semantic robustness and human evaluation further confirm its effectiveness in generating semantically reliable STL formulas.

5.2 RQ2: How effective is ClarifySTL at detecting vagueness and ambiguity in natural language requirements?

To answer RQ2, we evaluate the ability of ClarifySTL's Vagueness Detector and Ambiguity Detector to identify vague and ambiguous requirements. We compare ClarifySTL with closed-source LLM baselines, including GPT-4o, GPT-4o-mini, DeepSeek-V3, Gemini-2.5-Pro, and Claude-4-Sonnet, as well as non-expert users. The non-expert users are three computer science students who receive training on basic STL concepts and the NL-to-STL process, and are tasked with identifying vague or ambiguous requirements, and the prompting configuration are described in Section 4.5.

We evaluate the detectors on AmbiEval and on defective requirements identified from DeepSTL and STL-DivEn. To examine the effectiveness of triple-wise contrastive learning in the Ambiguity Detector, we further compare it with a pair-wise contrastive learning classifier and a fine-tuned LLaMA 3–8B classifier trained on Part B of AmbiEval.

Table 4. Vagueness Detection Performance of Models on AmbiEval, DeepSTL and STL-DivEn. (Accuracy, Precision, Recall, F1-score Expressed as %)

Model	AmbiEval				DeepSTL				STL-DivEn			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
GPT-4o	63.4	66.8	63.1	64.8	64.9	65.7	62.9	64.5	58.9	63.7	60.4	61.9
GPT-4o-mini	50.8	57.9	54.7	56.0	52.7	60.3	55.9	57.8	51.6	52.8	56.2	54.4
DeepSeek-V3	60.1	65.0	67.2	66.1	60.9	68.2	67.9	68.0	57.8	61.9	61.6	61.7
Gemini-2.5-Pro	67.6	65.2	62.4	63.8	70.9	65.4	62.8	64.1	66.9	60.8	59.9	60.3
Claude-4-Sonnet	67.1	68.7	69.8	69.2	72.6	73.4	70.2	71.8	65.8	66.8	70.4	68.6
Non-expert Users	79.3	79.0	80.1	79.5	82.4	81.8	82.6	82.2	80.1	80.0	80.9	80.4
Vagueness Detector	**91.1**	**90.2**	**91.8**	**91.0**	**92.2**	**91.4**	**91.9**	**91.6**	**89.1**	**90.0**	**90.9**	**90.4**

Table 5. Ambiguity Detection Performance of Models on AmbiEval, DeepSTL and STL-DivEn. (Accuracy, Precision, Recall, F1-score Expressed as %)

Model	AmbiEval				DeepSTL				STL-DivEn			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
GPT-4o	54.9	52.0	55.7	54.1	52.8	51.9	56.8	54.4	52.7	54.8	55.9	55.3
GPT-4o-mini	42.6	47.7	48.7	48.2	44.7	45.8	45.7	45.7	45.8	42.7	49.7	45.9
DeepSeek-V3	53.2	56.8	53.1	54.9	50.9	54.0	52.9	53.4	59.1	52.8	52.7	52.8
Gemini-2.5-Pro	51.2	52.9	52.3	52.6	57.1	55.8	54.9	55.3	54.0	58.0	55.1	56.5
Claude-4-Sonnet	54.4	53.1	54.1	53.6	53.9	55.0	55.2	55.1	57.1	55.8	57.6	56.7
Non-expert Users	72.0	71.8	72.4	72.1	72.9	72.8	73.4	73.1	71.9	72.9	73.3	73.1
Ambiguity Detector	**93.1**	**92.2**	**90.8**	**91.5**	**92.5**	**91.3**	**93.1**	**92.2**	**91.2**	**91.1**	**91.5**	**91.3**

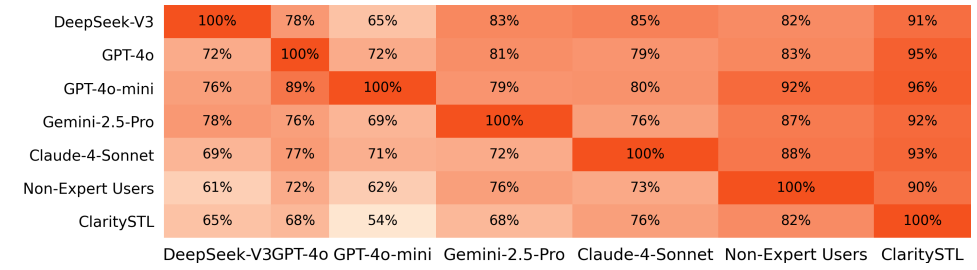

Fig. 8. Overlap of Vague Requirements Detected by Different Models.

5.2.1 Evaluation Results of Detection Tasks. The evaluation results of the vagueness detection task are presented in Table 4. The Vagueness Detector outperforms baselines across all datasets and metrics. For example, on the AmbiEval dataset, it attains an accuracy of 91.1%, a precision of 90.2%, a recall of 91.8%, and an F1-score of 91.0%. Notably, non-expert users also outperform LLMs that rely solely on intrinsic knowledge, although their performance remains lower than that of the Vagueness Detector.

The evaluation results of the ambiguity detection task are shown in Table 5. The Ambiguity Detector also achieves the best performance across all datasets and metrics compared with baselines. For instance, on the DeepSTL dataset, it reaches an accuracy of 92.5%, a precision of 91.3%, a recall of 93.1%, and an F1-score of 92.2%, clearly outperforming various LLMs. Similarly, non-expert users also perform better than closed-source LLMs.

These results demonstrate that both the Vagueness Detector and the Ambiguity Detector outperform models that rely solely on the intrinsic knowledge of LLMs and non-expert users in the detection tasks.

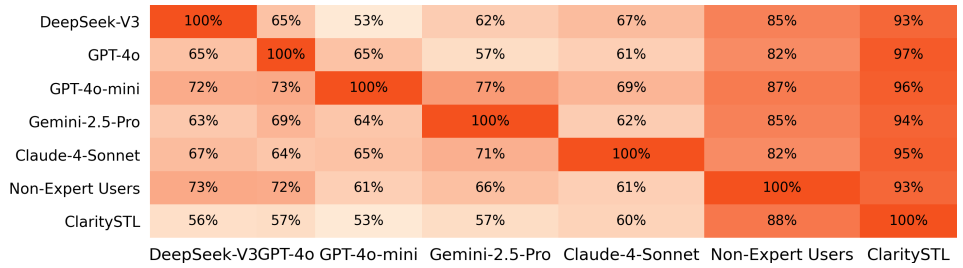

Fig. 9. Overlap of Ambiguous Requirements Detected by Different Models.

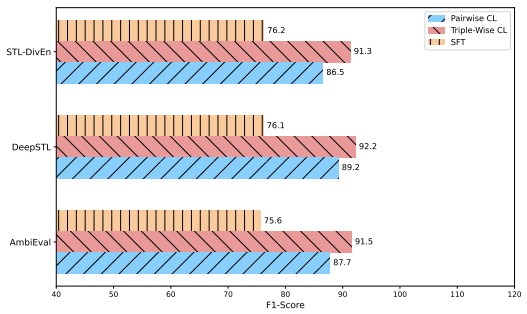

Fig. 10. Comparison of Different Ambiguity Identification Methods.

5.2.2 Overlapping Evaluation. As illustrated in Figure 8, each row represents the degree of overlap between the vague requirements correctly identified by one model and those identified by other models. Similarly, Figure 9 shows the overlap between the ambiguous requirements correctly recognized by different models. The color intensity of each rectangle deepens as the overlap increases, providing an intuitive visual cue for result interpretation. For example, 72% of the vague requirements correctly identified by GPT-4o overlap with those recognized by GPT-4o-mini (Row 2, Column 3 in Figure 8). As shown in the figures, the issues detected by other models are also recognized by our detectors, whereas the issues identified by our model are often missed by the others. This further confirms that our approach exhibits superior detection capability compared with the baselines.

5.2.3 Contrastive Learning in Ambiguity Detection. To validate the effectiveness of the contrastive learning module in the Ambiguity Detector, we compare three binary classification methods: triple-wise contrastive learning, pair-wise contrastive learning, and Supervised Fine-Tuning (SFT). As shown in Figure 10, triple-wise contrastive learning consistently outperforms the other two approaches. This is because, during training, triple-wise contrastive learning simultaneously models the relationships among anchor, positive, and negative samples. This enables the Ambiguity Detector to establish clearer semantic boundaries for ambiguous instances. In contrast, pair-wise contrastive learning only considers relationships between sample pairs, while SFT does not explicitly model the semantic associations among samples. Overall, these results demonstrate the advantage of triple-wise contrastive learning in capturing subtle semantic differences and achieving effective ambiguity detection.

Table 6. Vagueness Query Generation Abilities of Models on the AmbiEval Dataset.

Method		ROUGE	BERTScore	HumanEval	
				Correctness (%)	Clarity (%)
GPT-4o	default	0.386	0.679	63.8	71.6
	Ours	**0.591**	**0.875**	**92.6**	89.3
GPT-4o-mini	default	0.294	0.537	57.6	51.4
	Ours	0.486	0.731	86.8	82.4
DeepSeek-V3	default	0.395	0.608	62.0	62.3
	Ours	0.542	0.861	88.9	90.4
Gemini-2.5-Pro	default	0.360	0.631	69.0	68.8
	Ours	0.583	0.869	90.6	**92.1**
Claude-4-Sonnet	default	0.381	0.635	64.2	70.7
	Ours	0.574	0.831	89.1	90.5
Non-expert Users		0.545	0.713	71.6	90.1

Answer to RQ2. ClarifySTL effectively detects both vagueness and ambiguity in requirements. Its Vagueness Detector achieves **92.2%** accuracy on the DeepSTL dataset, **91.1%** on AmbiEval, and **89.1%** on STL-DivEn, with accuracy **9.8%**, **11.8%**, and **9.0%** higher than the best baseline, respectively. The Ambiguity Detector, leveraging triple-wise contrastive learning, reaches **93.1%** accuracy on AmbiEval, **92.5%** on DeepSTL, and **91.2%** on STL-DivEn, outperforming the best baseline by **21.1%**, **19.6%**, and **19.3%** in accuracy respectively.

5.3 RQ3: Are the queries generated by ClarifySTL effective and clear for users to provide clarification?

In this research question, we investigate the effectiveness and clarity of the clarification queries generated by ClarifySTL through a combination of metric-based and human evaluation on the AmbiEval dataset. Baseline models include GPT-4o, GPT-4o-mini, DeepSeek-V3, Gemini-2.5-Pro, Claude-4-Sonnet and non-expert users.

For human evaluation, three participants with at least two years of experience in STL specification writing are recruited, all of whom are graduate students in the CPS domain. Each participant receives a detailed evaluation guideline that defines the criteria for correctness (whether the query accurately identifies the vague or ambiguous part of the requirement) and clarity (whether the query is easy to understand and actionable for users). Each participant receives a set of vague or ambiguous requirements along with the corresponding generated queries. Each query is rated for correctness and clarity, with a score of 1 indicating satisfactory and 0 indicating unsatisfactory. The final metric is the percentage of queries receiving a score of 1. To measure inter-annotator agreement, we compute Fleiss' Kappa across the three annotators. The Fleiss' Kappa values are 0.89 for correctness and 0.85 for clarity, indicating agreement among the annotators.

For vagueness query generation, "default" refers to the few-shot prompting method, while "ours" denotes the CoT prompts we designed. For ambiguity query generation, "STL-based Analysis" indicates that LLMs analyze different STL formulas to generate queries, whereas "NL-based Analysis" follows the method proposed in Section 3.5, which analyzes natural language obtained through STL back-translation to generate queries.

Furthermore, we investigate the impact of the number of candidate STL formulas on the quality of queries in Ambiguity Inquirer. Specifically, the queries generated using 1 to 10 examples are evaluated by calculating their ROUGE and BERTScore with the ground truth, as well as their correctness and clarity (in the case of a single candidate STL formula, the LLM analyzes the differences between its back-translated natural language and the original requirement).

Table 7. Ambiguity Query Generation Abilities of Models on the AmbiEval Dataset.

Method		ROUGE	BERTScore	HumanEval	
				Correctness (%)	Clarity (%)
GPT-4o	default	0.281	0.428	45.6	70.6
	STL-based Analysis	0.401	0.648	67.1	85.6
	NL-based Analysis (Ours)	**0.637**	**0.894**	**94.6**	90.6
GPT-4o-mini	default	0.194	0.388	41.7	52.0
	STL-based Analysis	0.322	0.590	60.5	77.9
	NL-based Analysis (Ours)	0.486	0.785	90.7	87.6
DeepSeek-V3	default	0.296	0.442	46.0	61.2
	STL-based Analysis	0.427	0.661	67.6	82.6
	NL-based Analysis (Ours)	0.616	**0.894**	**94.6**	**91.4**
Gemini-2.5-Pro	default	0.259	0.414	43.6	67.6
	STL-based Analysis	0.376	0.629	66.7	79.8
	NL-based Analysis (Ours)	0.629	0.887	94.1	90.1
Claude-4-Sonnet	default	0.287	0.456	47.5	71.9
	STL-based Analysis	0.419	0.670	68.6	79.1
	NL-based Analysis (Ours)	0.634	0.884	93.6	88.6
Non-expert Users		0.496	0.731	76.6	86.2

5.3.1 Evaluation of Query Generation. Table 6 presents the evaluation results of Vagueness Inquirer in generating queries for vagueness issues. Overall, Vagueness Inquirer demonstrates effectiveness and clarity in generating queries. For metric-based evaluation, the GPT-4o-based Vagueness Inquirer achieves the best performance, with ROUGE and BERTScore values of 0.591 and 0.875 respectively. In human evaluation, it attains the highest correctness at 92.6%, while the Gemini-2.5-Pro-based Vagueness Inquirer achieves the highest clarity at 92.1%. Meanwhile, the queries generated by non-expert users are of relatively high quality, outperforming all default baselines but still underperforming our method.

Table 7 presents the evaluation results of the Ambiguity Inquirer in generating queries for ambiguity issues. For metric-based evaluation, the GPT-4o-based Ambiguity Inquirer performs best, achieving a ROUGE score of 0.637 and a BERTScore of 0.894, which are comparable to the scores of DeepSeek-V3. In human evaluation, both GPT-4o and DeepSeek-V3 attain the highest correctness at 94.6%, while DeepSeek-V3 achieves the highest clarity at 91.4%. In comparison, the STL-based analysis shows a noticeable drop in both metric-based evaluation and human evaluation, performing slightly better than the default baselines but lower than non-expert users and our method. This indicates that leveraging LLMs for natural language analysis is more effective than directly analyzing STL formulas.

5.3.2 Impact of the Number of Candidate STL Formulas. As illustrated in Figure 11, the number of candidate STL formulas used in prompts impacts model performance. When the number of candidate STL formulas is fewer than three, increasing their quantity enhances the model's ability to generate high-quality queries. However, when the number exceeds three, the performance improvement becomes marginal, and excessive candidates even lead to a decline. This occurs because adding a few candidates provides helpful contextual cues, but too many introduce noise and redundancy, making it harder for the model to focus and thus reducing query generation quality.

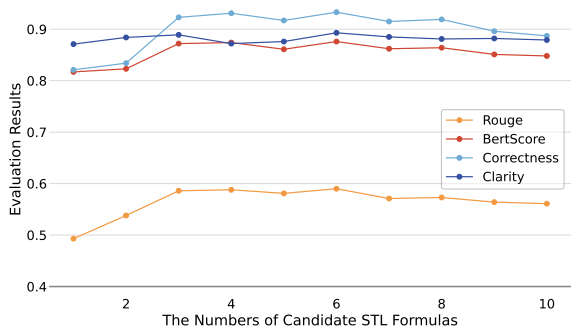

Fig. 11. Impact of the Number of Candidate STL Formulas.

Answer to RQ3. ClarifySTL proves effective and precise in generating clarification queries. The Vagueness Inquirer achieves a ROUGE score of **0.591** and a BERTScore of **0.875**, while the Ambiguity Inquirer scores **0.637** and **0.894**, respectively. Human evaluation confirms the quality, with the Vagueness Inquirer reaching **92.6%** correctness and **92.1%** clarity, and the Ambiguity Inquirer achieving **94.6%** correctness and **91.4%** clarity, outperforming the baselines.

6 Discussion

6.1 Ablation Study

Table 8. Ablation Experiments Conducted on the DeepSTL Dataset. (✔ indicates including the component, ✖ indicates removing the component. Acc. stands for accuracy, Avg. stands for average, and Sem. Rob. stands for semantic robustness.)

Vagueness Stage		Ambiguity Stage		Formula Acc. (%)	Template Acc. (%)	BLEU	Sem. Rob. (%)	Avg. Iteration Rounds per Requirement
Detector	Inquirer	Detector	Inquirer					
✔		✔		67.12	70.03	0.5736	82.7	3.7
✔		✖		54.82	53.91	0.4386	69.4	3.2
✖		✔		53.76	54.83	0.4471	68.7	2.9
✔	✖	✔		62.18	63.86	0.5218	77.9	6.5
✔		✔	✖	61.27	61.93	0.5094	76.8	6.0
✖	✔	✔		58.91	57.84	0.4892	72.6	2.3
✔		✖	✔	56.84	57.35	0.4784	71.5	2.9

The results of the ablation experiments are shown in Table 8. Removing the Detector or Inquirer means using only the few-shot prompt to enable LLMs to detect errors or generate queries, respectively. It can be observed that when the Inquirer is removed, the number of iterations per requirement increases, indicating higher labor costs. Meanwhile, the metric-based evaluation metrics decrease, which suggests that the lack of the Inquirer partially causes annotators to provide inaccurate clarification responses. When the Detector is removed, the automated metrics decline and the average number of iteration decreases. This indicates that fewer errors are detected, resulting in the early termination of the clarification process. Moreover, the removal of either stage leads to a drop in metric-based evaluation, demonstrating the significance of proposing both stages.

6.2 Rationale for Selecting the Base Model

The results in Table 9 demonstrate that the performance of the detection agents is influenced by the choice of the underlying base model. Models with larger parameter sizes consistently outperform

Table 9. Impact of Different Base Models on the AmbiEval Dataset for Vagueness and Ambiguity Detection.

Base Model	Vagueness Detector				Ambiguity Detector			
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LLaMA 3-8B	91.1	90.2	91.8	91.0	93.1	92.2	90.8	91.5
LLaMA 3-14B	**94.0**	92.6	**93.8**	**93.2**	**94.9**	**93.1**	92.8	92.9
Qwen 3-8B	90.0	88.9	87.8	88.3	90.9	89.8	89.2	89.5
Qwen 3-14B	92.9	**92.8**	92.6	92.7	93.9	92.7	**93.2**	**92.9**

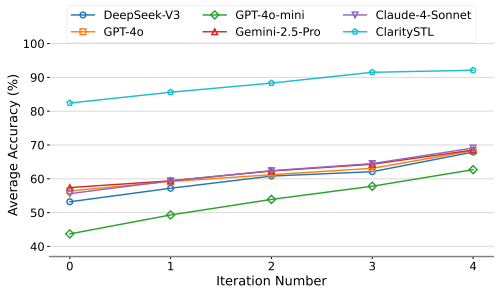

Fig. 12. Impact of Iterations on Vagueness Clarification in DeepSTL.

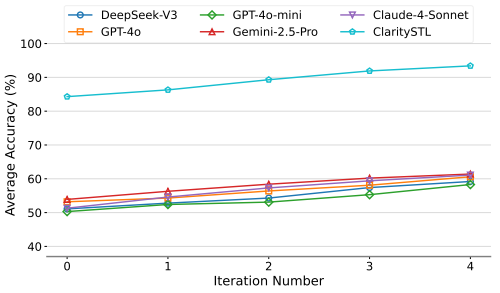

Fig. 13. Impact of Iterations on Ambiguity Clarification in DeepSTL.

their smaller counterparts in both vagueness and ambiguity detection. Specifically, LLaMA 3-14B achieves 94.0% accuracy and 93.2% F1-score for vagueness detection, as well as 94.9% accuracy for ambiguity detection. Qwen 3-14B also performs competitively, attaining an F1-score of 92.9% in ambiguity detection. In contrast, smaller models such as LLaMA 3-8B and Qwen 3-8B show drops in all metrics. However, considering the trade-off between performance, computational efficiency, and deployment scalability, we adopt LLaMA 3-8B as the primary base model for experiments.

As shown in Table 6 and Table 7, GPT-4o achieves outstanding performance across most metrics in both tasks. Therefore, we select GPT-4o as the base model for the Inquirer, considering its strong capability and reasonable computational cost.

6.3 Impact of Iteration Numbers

To investigate the effect of the number of iterations, we measure the transformation efficiency of different models in clarifying vagueness and ambiguity on STL-DivEn as the number of iterations increases. In our experiments, an iteration value of 0 represents the initial clarification. Additionally, the iteration number indicates the maximum number of iterations, even if the requirement still contains vagueness or ambiguity, no further iterations will be performed. As shown in Figures 12 and 13, the iterative structure has a positive impact on the clarification process. For example, when the iteration count is 0, ClarifySTL achieves an accuracy of 84.3%. As the number of iterations increases, the information in the requirements is progressively supplemented or clarified, resulting in more accurate natural language requirements and, consequently, better STL formula generation. This further demonstrates that providing clearer natural language input effectively improves the accuracy of the natural language to STL transformation.

6.4 Reliability of Ambiguity Inquirer

The Ambiguity Inquirer involves multiple steps, including candidate STL sampling, templated back-translation, GPT-based difference analysis, and query generation. Since these steps may introduce

Table 10. Back-Translation Fidelity of Candidate STL Formulas. Acc. stands for accuracy.

Method	Atomic Propositions Acc. (%)	Temporal Operator Acc. (%)	Numerical Information Acc. (%)	Boolean Structure Acc. (%)	Semantic Consistency (%)
GPT-4o	95.6	94.8	94.1	93.9	95.2
GPT-4o-mini	91.2	90.4	89.7	89.4	90.8
DeepSeek-V3	95.1	94.3	93.8	93.5	94.9
Gemini-2.5-Pro	94.7	94.0	93.4	93.0	94.5
Claude-4-Sonnet	94.3	93.7	93.1	92.8	94.2

Table 11. Query Quality under Erroneous Candidate Settings.

Candidates Setting	ROUGE	BERTScore	Correctness (%)	Clarity (%)
Reasonable	0.641	0.897	95.1	91.8
Reasonable + Erroneous	0.635	0.891	94.3	91.0

uncertainty, we further evaluate the reliability of this pipeline from five aspects: back-translation fidelity, the influence of erroneous candidates, spurious ambiguity on unambiguous requirements, stability under repeated sampling and different temperatures, and consistency preservation after clarification. For the human evaluations in this subsection, we hire two annotators with STL experience to conduct the following assessments.

6.4.1 Back-Translation Fidelity. To examine whether templated back-translation faithfully expresses candidate STL formulas in natural language, we conduct a human evaluation on pairs of candidate STL formulas and their back-translated descriptions. The annotators assess whether each description correctly preserves the atomic propositions, temporal operators, numerical information, Boolean structure, and overall semantics of the corresponding STL formula. A description is counted as correct only when both annotators agree.

As shown in Table 10, all models achieve high fidelity across the five aspects. GPT-4o performs best, with scores between 93.9% and 95.6%, while the other models also maintain generally high accuracy. These results indicate that the back-translated descriptions can provide reliable input for the subsequent difference analysis.

6.4.2 Impact of Erroneous Candidates. We investigate whether erroneous candidates produced during sampling affect ambiguity clarification. Annotators divide the sampled candidate STL formulas and their back-translated descriptions into reasonable interpretations and erroneous candidates, where the latter are caused by translation or generation errors. We then compare query quality when using only reasonable candidates and when mixing reasonable and erroneous candidates.

As shown in Table 11, the two settings achieve very close results. Adding erroneous candidates only slightly decreases ROUGE from 0.641 to 0.635, BERTScore from 0.897 to 0.891, correctness from 95.1% to 94.3%, and clarity from 91.8% to 91.0%. This suggests that occasional erroneous candidates do not substantially affect ambiguity clarification, because genuine ambiguity points tend to form stable divergence patterns across reasonable candidates, while erroneous candidates usually appear as isolated noise.

6.4.3 Spurious Ambiguity Analysis on Unambiguous Requirements. We conduct a sanity-check experiment to examine whether the ambiguity pipeline incorrectly triggers clarification for unambiguous requirements. Specifically, we feed 50 unambiguous requirements into the Ambiguity Inquirer and execute the full pipeline. As shown in Table 12, most unambiguous requirements are correctly handled by the Ambiguity Inquirer without generating clarification queries. Across the five LLM backbones, the majority of outputs are categorized as no-ambiguity outputs, while only a small number of cases lead to clarification queries. This indicates that the pipeline does not simply

Table 12. Sanity-Check Results of the Ambiguity Inquirer on 50 Unambiguous Requirements.

Model	# No-Ambiguity Outputs	# Clarification Queries	# Uncertain/Invalid Outputs
GPT-4o	48	0	2
GPT-4o-mini	42	2	6
DeepSeek-V3	47	1	2
Gemini-2.5-Pro	45	0	5
Claude-4-Sonnet	46	0	4

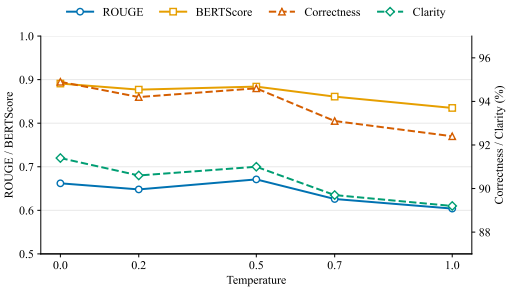

Fig. 14. Temperature Sensitivity of the Ambiguity Inquirer.

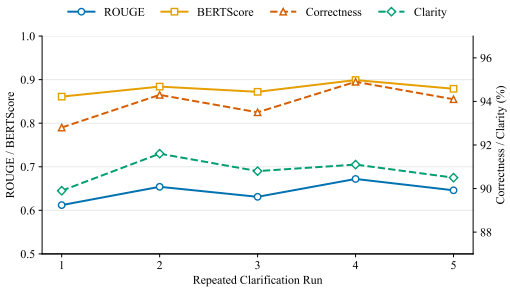

Fig. 15. Performance of the Ambiguity Inquirer under Repeated Clarification Procedure.

Table 13. Alignment Preservation Between Refined and Original Requirements in Ambiguity Clarification.

Model	With Guardrails Prompts		Without Guardrails Prompts	
	Requirement Consistency Rate (%)	Avg. Clarification Rounds	Requirement Consistency Rate (%)	Avg. Clarification Rounds
GPT-4o	99.2	3.6	98.1	3.8
GPT-4o-mini	97.8	4.2	95.9	4.4
DeepSeek-V3	99.0	3.7	98.0	3.9
Gemini-2.5-Pro	98.6	3.8	97.2	4.0
Claude-4-Sonnet	98.4	3.8	97.1	4.0

treat superficial differences among sampled STL candidates as evidence of ambiguity. Instead, it generally refrains from asking users for clarification when no concrete ambiguous segment can be localized in the original requirement.

6.4.4 Stability and Temperature Sensitivity. To evaluate the influence of candidate sampling and temperature settings, we repeat the ambiguity clarification pipeline for the same ambiguous requirements under multiple runs and different temperatures. Annotators judge whether the generated queries capture the real ambiguity in each requirement. As shown in Figures 14 and 15, the query correctness remains stable across repeated runs and stays high under different temperature settings. This indicates that the ambiguity clarification pipeline does not rely on a single accidental sampling result and is not highly sensitive to the temperature used in candidate STL generation.

6.4.5 Consistency Preservation of Refined Requirements. We further examine whether the multi-step LLM orchestration in ClarifySTL resolves ambiguity while preserving alignment with the original requirement. For each LLM backbone, we sample 50 ambiguous requirements from AmbiEval and ask annotators to evaluate whether the refined requirement correctly incorporates the user's clarification while preserving the non-ambiguous parts of the original requirement. Annotators judge whether the refined requirement preserves the original content, modifies only the user-clarified ambiguous fragment, and avoids introducing constraints not confirmed by the user. A

refined requirement is counted as requirement-consistent only when all three conditions are satisfied, and the Requirement Consistency Rate is computed as the percentage of such cases.

As shown in Table 13, the consistency rate remains high across all LLM backbones, and the setting with guardrails slightly improves the rate compared with the setting without guardrails. Meanwhile, the average clarification rounds remain close under the two settings, indicating that the guardrails improve alignment preservation without increasing the user interaction burden. These results suggest that ClarifySTL's ambiguity clarification stage is not an unconstrained rewriting process, but generally leads refinement to the user-confirmed ambiguous fragment while preserving the original natural-language specification.

6.5 Threats to Validity

In this subsection, we discuss the main threats to the validity of ClarifySTL as follows:

Threats to external validity. The primary threat to the external validity of this study lies in the quality of the experimental subjects. It remains uncertain to what extent the improvements achieved by ClarifySTL can be generalized to other STL transformation benchmarks. To mitigate this concern, we employ the DeepSTL and STL-DivEn datasets, consistent with prior studies. We also introduce specialized benchmark datasets for vagueness and ambiguity detection as well as query generation. This approach enhances the diversity and coverage of our evaluation. Furthermore, the evaluation results could be influenced by the participants' particular experience and background. To minimize this effect, we provide a standardized evaluation procedure and detailed guidelines to ensure consistency and reproducibility. In future work, we plan to expand the participant pool to further assess the generalizability of ClarifySTL across a broader range of user backgrounds.

Given the limited availability of openly accessible industrial STL specifications, AmbiEval is constructed from DeepSTL and STL-DivEn, two public NL-to-STL benchmarks designed to capture representative STL usage characteristics. In particular, DeepSTL samples STL formulas according to empirical operator distributions, and STL-DivEn is also constructed to reflect similar STL usage patterns. Building on these benchmarks, AmbiEval is aligned with the practical STL operator distribution and expression patterns reflected in DeepSTL and STL-DivEn. Beyond the dataset construction, the predefined vagueness categories in ClarifySTL are derived from the information required by STL formalization, including temporal intervals, numerical thresholds, and conditional logic. Although informal or domain-specific requirements may differ in surface expressions, their underlying underspecified information often still corresponds to these STL-relevant elements. Therefore, the prompt-based Inquirer can generalize to requirements whose vagueness falls within these STL-grounded categories. Nevertheless, requirements involving highly domain-specific assumptions or defect types may require additional category design or domain adaptation, which we will address in future work.

Threats to internal validity. LLMs are highly sensitive to prompts and hyperparameters, especially the number of task examples and natural language instructions, which can significantly affect model performance. To minimize such variability, we conduct an experimental analysis on the number of examples used in ClarifySTL and select the optimal number in all experimental settings. We also maintain consistent prompts and hyperparameter settings in the baseline models. Another potential threat is that during the process of generating queries for ambiguity, we rely on GPT-4o to generate candidate STL formulas. According to Table 2, this process may introduce new errors, which are in fact identified by the framework as ambiguities requiring clarification. When additional information is received from the user, the quality of the STL formulas generated by GPT-4o improves, eventually eliminating these errors. This also demonstrates the effectiveness of our approach: the clearer the input natural language requirement, the more accurate the generated STL formulas.

A potential concern is whether multiple vagueness issues coexisting in a single requirement may cause context loss during clarification. As described in Section 3, the Vagueness Detector and Inquirer operate iteratively. In each round, the Detector identifies the remaining vagueness in the current requirement, the Inquirer generates a targeted query, and the refined requirement is sent back to the Detector for the next round, while the original requirement is preserved as the main context across rounds. The effectiveness of this iterative mechanism is further empirically supported in Section 6.3, which shows that the clarification quality improves as the number of iterations increases. Therefore, multiple coexisting vagueness issues can be progressively resolved through the iterative interaction rather than handled in a single round.

Threats to construct validity. This paper uses Formula Accuracy, Template Accuracy, and BLEU to evaluate the accuracy of the generated STL formulas. Although these metrics cannot fully replace human judgment, they provide a rigorous and objective quantitative measure, enabling rapid assessment of model performance. In addition, we conduct human evaluations to verify the results produced by the model across different types of natural language requirement correction tasks. For the evaluation of query generation quality in Section 5.3, we adopt the widely used ROUGE and BERTScore metrics from text generation tasks to ensure fair comparisons. Future work will incorporate additional human evaluations to further validate model performance. Moreover, ClarifySTL focuses on leveraging agents to address vagueness or ambiguity in single-sentence natural language requirements. In the future, we plan to extend the design components of ClarifySTL to documents containing multiple requirement statements and apply these extended components to a broader range of scenarios.

7 Related Work

In this paper, we propose a new model for transforming natural language into signal temporal logic, which is an extension of temporal logic. Thus, our work mainly relates to two research areas: (i) from NL to TL, and (ii) from NL to STL. In this section, we summarize related work in these two areas.

7.1 From Natural Language to Temporal Logic

Some works have investigated the transformation of natural language into Temporal Logic specifications [3, 4, 12, 21, 40, 50, 56]. For instance, a catalog of temporal logic formulas that capture common specification patterns in the design of concurrent and reactive systems was proposed in [11]. Controlled English has also been transformed into TL through the use of syntactic and grammatical dependency parsing, together with predefined mapping rules [49, 61]. Santos et al. [49], Žilka [61] transform controlled English into LTL by applying syntactic and grammatical dependency parsing combined with predefined mapping rules. Similarly, the ARSENAL framework [23] uses NLP techniques such as n-gram analysis and dependency parsing to transform natural language requirements into formal specifications, including TL. Although these methods are effective in specific domains, they rely on handcrafted rules and restricted language inputs, which limit their ability to handle more diverse and complex expressions. To overcome these limitations, the nl2spec method [9] combines human feedback with LLMs' ability to generate LTL formulas, which relies heavily on the user's professional knowledge. In addition, NL2TL [6] mitigates the reliance on rule design by fine-tuning a T5 model on NL–TL pairs generated by LLMs. However, these TL-focused methods are not readily extended to STL, because STL introduces real-valued signals and continuous-time constraints, which pose challenges not typically addressed by traditional TL.

7.2 From Natural Language to Signal Temporal Logic

As an extension of TL that incorporates real-valued dense-time signals, STL has gained widespread use in both academia and industry to meet the requirements of CPS [5, 36, 39]. Consequently, numerous efforts have been made to transform natural language into STL [6, 19, 25, 31, 45]. For example, DeepSTL [25] trains a Transformer model using grammar-based synthetic data. Although this approach ensures formal consistency, it heavily relies on handcrafted rules and artificial data, which fail to capture the diversity in real-world natural language. As a result, its generalization capability is limited when applied to open-domain inputs. To facilitate broader evaluation, STL-DivEn [19] has been introduced for NL-to-STL translation. This dataset provides diverse linguistic expressions and complex signal patterns, enabling standardized assessment of model robustness and semantic fidelity. KGST [19] uses a generate-then-refine approach by first fine-tuning LLMs to generate initial STL formulas, then refining them with external knowledge. Despite their impressive performance, DeepSTL and KGST, as supervised learning approaches, rely on fixed training objectives for supervision. This results in limited mechanisms for detecting or resolving ambiguities in natural language input. Consequently, when the input contains vague or ambiguous information, these approaches struggle to accurately capture the intended semantics. Dialogue-based approaches [41] mitigate this issue by leveraging the internal knowledge of LLMs to engage in dialogue with users regarding unclear parts, refining requirements for STL generation. However, the internal knowledge of LLMs is typically limited, making it difficult for them to identify vague or ambiguous information expressed in requirements and accurately generate queries to interact with users. The objective of ClarifySTL is to provide a novel interactive framework: we inject vagueness classification knowledge through fine-tuning and design CoT-based prompts to clarify vagueness. Additionally, we capture the subtle differences between ambiguous and unambiguous sentences through contrastive learning, and analyze the multiple potential interpretations of the same input requirement to resolve ambiguity. Experimental results demonstrate that the proposed detection and query generation methods enable ClarifySTL to conduct efficient interactions and requirement clarifications, generating STL specifications that accurately reflect user intent.

8 Conclusion and Future Work

This paper introduces an interactive NL–to-STL transformation framework that guides users in clarifying vagueness and ambiguity in natural language requirements, helping users transform such requirements into STL formulas that align with their intended semantics. ClarifySTL is built upon LLMs fine-tuned with AmbiEval and GPT-4o, enabling it to accurately identify vague or ambiguous requirements and generate explicit queries. Metric-based evaluations on the DeepSTL, STL-DivEn, and AmbiEval benchmarks demonstrate that ClarifySTL outperforms baselines. Human evaluations further validate the effectiveness of our framework, showing that it effectively guides users in clarification. We also conduct dedicated experiments to evaluate the accuracy of our vagueness and ambiguity detection agents, as well as the correctness and clarity of the corresponding generated queries. In future work, we plan to explore cross-domain adaptation and extend ClarifySTL to document-level STL transformation tasks to further enhance its applicability.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62192731, 62192732, 62192730, 92582203, and W2511064; the CAS Project for Young Scientists in Basic Research under Grant No. YSBR-123; and ISCAS Basic Research under Grant No. ISCAS-JCZD-202406.

References

[1] Daniel M Berry and Erik Kamsties. 2004. Ambiguity in Requirements Specification. In *Perspectives on software requirements*. Springer, 7–44.

[2] Daniel M Berry, Erik Kamsties, and Michael M Krieger. 2003. From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity. *Citeseer* (2003).

[3] Marco Bombieri, Daniele Meli, Diego Dall'Alba, Marco Rospocher, and Paolo Fiorini. 2023. Mapping natural language procedures descriptions to linear temporal logic templates: an application in the surgical robotic domain. *Applied Intelligence* 53, 22 (2023), 26351–26363.

[4] Igor Buzhinsky. 2019. Formalization of natural language requirements into temporal logics: a survey. In *2019 IEEE 17th international conference on industrial informatics (INDIN)*, Vol. 1. IEEE, 400–406.

[5] Hongkai Chen, Scott A Smolka, Nicola Paoletti, and Shan Lin. 2023. An STL-based approach to resilient control for cyber-physical systems. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*. 1–12.

[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, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 15880–15903.

[7] Yongchao Chen, Yilun Hao, Yueying Liu, Yang Zhang, and Chuchu Fan. [n. d.]. CodeSteer: Symbolic-Augmented Language Models via Code/Text Guidance. In *Forty-second International Conference on Machine Learning*.

[8] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[9] 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 *CAV 2023 (LNCS, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 383–396.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[11] 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*. 411–420.

[12] William H English, Dominic Simon, Sumit Kumar Jha, and Rickard Ewetz. [n. d.]. Grammar-Forced Translation of Natural Language to Temporal Logic using LLMs. In *Forty-second International Conference on Machine Learning*.

[13] Gidon Ernst, Paolo Arcaini, Ismail Bennani, Aniruddh Chandratre, Alexandre Donzé, Georgios Fainekos, Goran Frehse, Khouloud Gaaloul, Jun Inoue, Tanmay Khandait, Logan Mathesen, Claudio Menghi, Giulia Pedrielli, Marc Pouzet, Masaki Waga, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. 2021. ARCH-COMP 2021 Category Report: Falsification with Validation of Results. In *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21) (EPiC Series in Computing, Vol. 80)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 133–152. doi:10.29007/xwl1

[14] Gidon Ernst, Paolo Arcaini, Ismail Bennani, Alexandre Donzé, Georgios Fainekos, Goran Frehse, Logan Mathesen, Claudio Menghi, Giulia Pedrielli, Marc Pouzet, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. 2020. ARCH-COMP 2020 Category Report: Falsification. In *7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20) (EPiC Series in Computing, Vol. 74)*. 140–152.

[15] Gidon Ernst, Paolo Arcaini, Georgios Fainekos, Federico Formica, Jun Inoue, Tanmay Khandait, Mohammad Mahdi Mahboob, Claudio Menghi, Giulia Pedrielli, Masaki Waga, Yoriyuki Yamagata, and Zhenya Zhang. 2022. ARCH-COMP 2022 Category Report: Falsification with Ubounded Resources. In *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22) (EPiC Series in Computing, Vol. 90)*, Goran Frehse, Matthias Althoff, Erwin Schoitsch, and Jeremie Guiochet (Eds.). EasyChair, 204–221. doi:10.29007/fhnk

[16] Saad Ezzini, Sallam Abualhaija, Chetan Arora, Mehrdad Sabetzadeh, and Lionel Briand. 2021. MAANA: An Automated Tool for DoMAin-Specific HANdling of Ambiguity. In *2021 IEEE/ACM 43rd ICSE-Companion*. 188–189. doi:10.1109/ICSE-Companion52605.2021.00082

[17] Saad Ezzini, Sallam Abualhaija, Chetan Arora, Mehrdad Sabetzadeh, and Lionel C Briand. 2021. Using Domain-Specific Corpora for Improved Handling of Ambiguity in Requirements. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1485–1497.

[18] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* (2024).

[19] 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. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 10446–10458. https://aclanthology.org/2025.findings-acl.544/

[20] Joseph L. Fleiss. 1971. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin* 76, 5 (1971), 378–382.

[21] 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 AAAI Conference on Artificial Intelligence*, Vol. 37. 16428–16430.

[22] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics (ACL), 6894–6910.

[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] Yann Gilpin, Vince Kurtz, and Hai Lin. 2020. A smooth robustness measure of signal temporal logic for symbolic control. *IEEE Control Systems Letters* 5, 1 (2020), 241–246.

[25] Jie He, Ezio Bartocci, Dejan Nickovic, Haris Isakovic, and Radu Grosu. 2022. DeepSTL - From English Requirements to Signal Temporal Logic. In *ICSE 2022*. ACM, 610–622.

[26] Elad Hoffer and Nir Ailon. 2015. Deep Metric Learning Using Triplet Network. In *International Workshop on Similarity-based Pattern Recognition*.

[27] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[28] Erik Kamsties and Barbara Peach. 2000. Taming ambiguity in natural language requirements. In *Proceedings of the Thirteenth International Conference on Software and Systems Engineering and Applications*.

[29] Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[30] Dhanashree Kulkarni, Andrew N Fisher, and Chris J Myers. 2013. A new assertion property language for analog/mixed-signal circuits. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. IEEE, 1–8.

[31] Danyang Li, Mingyu Cai, Cristian-Ioan Vasile, and Roberto Tron. 2023. Learning Signal Temporal Logic through Neural Network for Interpretable Classification. In *2023 American Control Conference (ACC)*. IEEE, 1907–1914.

[32] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell P. Marcus, and Hadas Kress-Gazit. 2015. Provably correct reactive control from natural language. *Auton. Robots* 38, 1 (2015), 89–105.

[33] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[34] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[35] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. 2024. Improving requirements completeness: Automated assistance through large language models. *Requirements Engineering* 29, 1 (2024), 73–95.

[36] Curtis Madsen, Prashant Vaidyanathan, Sadra Sadraddini, Cristian Ioan Vasile, Nicholas A. DeLateur, Ron Weiss, Douglas Densmore, and Calin Belta. 2018. Metrics for Signal Temporal Logic Formulae. In *57th IEEE Conference on Decision and Control, CDC 2018, Miami, FL, USA, December 17-19, 2018*. IEEE, 1542–1547.

[37] Sebastian Maierhofer, Anna-Katharina Rettinger, Eva Charlotte Mayer, and Matthias Althoff. 2020. Formalization of interstate traffic rules in temporal logic. In *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 752–759.

[38] Oded Maler and Dejan Ničković. 2004. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT 2004*. LNCS, Vol. 3253. Springer, 152–166. doi:10.1007/978-3-540-30206-3_12

[39] Yuchen Mao, Tianci Zhang, Xu Cao, Zhongyao Chen, Xinkai Liang, Bochen Xu, and Hao Fang. 2024. Nl2stl: Transformation from logic natural language to signal temporal logics using llama2. In *2024 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE International Conference on Robotics, Automation and Mechatronics (RAM)*. IEEE, 469–474.

[40] 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.

[41] Sara Mohammadinejad, Sheryl Paul, Yuan Xia, Vidisha Kudalkar, Jesse Thomason, and Jyotirmoy V Deshmukh. 2024. Systematic translation from natural language robot task descriptions to stl. In *International Conference on Bridging the Gap between AI and Reality*. Springer, 259–276.

[42] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[44] A Paszke. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[45] Roma Patel, Roma Pavlick, and Stefanie Tellex. 2019. Learning to ground language to temporal logical form. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

[46] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS 1977*. IEEE, 46–57. doi:10.1109/SFCS.1977.32

[47] KC Pragyan, Rambod Ghandiparsi, Thomas Herron, John Heaps, and Mitra Bokaei Hosseini. 2025. Demystifying Feature Requests: Leveraging LLMs to Refine Feature Requests in Open-Source Software. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*. IEEE, 104–116.

[48] Vasumathi Raman, Constantine Lignos, Cameron Finucane, Kenton CT Lee, Mitchell P Marcus, and Hadas Kress-Gazit. 2013. Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language.. In *Robotics: science and systems*, Vol. 2. Berlin, Germany, 2–1.

[49] Tainã Santos, Gustavo Carvalho, and Augusto Sampaio. 2018. Formal modelling of environment restrictions from natural-language requirements. In *SBMF 2018*. Springer, 252–270.

[50] Eshita Shukla, Quinn Thibeault, and Giulia Pedrielli. 2025. A gray box approach for Large Language Model-guided Natural Language to Temporal Logic Automatic Translation. In *Proceedings of the ACM/IEEE 16th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2025)*. 1–2.

[51] Stefanie Tellex, Nakul Gopalan, Hadas Kress-Gazit, and Cynthia Matuszek. 2020. Robots that use language. *Annual Review of Control, Robotics, and Autonomous Systems* 3, 1 (2020), 25–55.

[52] JG Thistle and WM Wonham. 1986. Control problems in a temporal logic framework. *Internat. J. Control* 44, 4 (1986), 943–976.

[53] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[54] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.

[55] Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–31.

[56] 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)*. IEEE, 204–215.

[57] Ruixuan Yan, Agung Julius, Maria Chang, Achille Fokoue, Tengfei Ma, and Rosario Uceda-Sosa. 2021. Stone: Signal temporal logic neural network for time series classification. In *2021 International Conference on Data Mining Workshops (ICDMW)*. IEEE, 778–787.

[58] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[59] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. BERTScore: Evaluating Text Generation with BERT. In *International Conference on Learning Representations*.

[60] Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. 400–410.

[61] Lukáš Žilka. 2010. *Temporal logic for man.* Ph. D. Dissertation. Master's thesis, Brno University of Technology.

Received 8 November 2025; revised 30 April 2026; accepted 10 August 2026