DeepSTL - From English Requirements to Signal Temporal Logic

Anonymous Author(s)

ABSTRACT

Formal methods provide very powerful tools and techniques for the design and analysis of complex systems. Their practical application remains however limited, due to the widely accepted belief that formal methods require extensive expertise and a steep learning curve. Writing correct formal specifications in form of logical formulas is still considered to be a difficult and error prone task.

In this paper we propose DeepSTL, a tool and technique for the translation of informal requirements, given as free English sentences, into Signal Temporal Logic (STL), a formal specification language for cyber-physical systems, used both by academia and advanced research labs in industry. A major challenge to devise such a translator is the lack of publicly available informal requirements and formal specifications. We propose a two-step workflow to address this challenge. We first design a grammar-based generation technique of synthetic data, where each output is a random STL formula and its associated set of possible English translations. In the second step, we use a state-of- the-art transformer-based neural translation technique, to train an accurate attentional translator of English to STL. The experimental results show high translation quality for patterns of English requirements that have been well trained, making this workflow promising to be extended for processing more complex translation tasks.

ACM Reference Format:

Anonymous Author(s). 2022. DeepSTL - From English Requirements to Signal Temporal Logic. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

"What is reasonable is real. That which is real is reasonable". This famous proposition from Hegel, saying that everything has its "logic", often resonates in Alice's mind. Alice is a verification engineer responsible for safety-critical cyber-physical systems (CPS). She advocates the use of formal methods with requirements specified in logic, as part the development of complex CPS.

Formal specifications enable rigorous reasoning about a CPS product (for example its model checking or systematic testing) during all its design phases, as well as during operation (for example via runtime verification) [1]. Alice is frustrated by the resistance of her colleagues to adopt formal methods in their design methodology. She is aware that one major bottleneck in a wider acceptance of these techniques results from the steep learning curve to translate informal requirements expressed in natural language into formal specification. The correspondence between a requirement written in English and its temporal logic formalization is not always straightforward, as illustrated in the example below:

- **English Requirement**:
  Whenever V_Mot is detected to become equal to 0, then at a time point starting after at most 100 time units Spd_Act shall continuously remain on 0 for at least 20 time units.
- **Signal Temporal Logic (STL)**:
  $\mathbf{G}(\mathbf{rise}(\text{V\_Mot} = 0) \rightarrow \mathbf{F}_{[0,100]}\mathbf{G}_{[0,20]}\ (\text{Spd\_Act} = 0))$

Bob is Alice's colleague and an expert in machine learning. He introduces Alice to the tremendous achievements in natural language processing (NLP), demonstrated by applications such as Google Translate and DeepL. Alice is impressed by the quality of translations between natural languages. She realizes that NLP is a technology that can reduce the gap between engineers and formal methods, and significantly increase the acceptance of rigorous specifications.

However, Alice also observes that this potential solution does not come without challenges. In order to build a translator from one spoken language to another, there is a huge amount of text available in both languages that can be used for training and there is also a series of systematic translation solutions. In contrast, for translating CPS requirements given in natural language into formal specifications, there are two major challenges:

- **Challenge 1**: Lack of available training data. The informal requirement documents are sparse and often not publicly available, and formal specifications are even sparser.
- **Challenge 2**: No mature solutions for translating English requirements into formal specifications, where special features of these two languages need to be considered.

In this paper, as a first attempt to adopt NLP to tackle the above two challenges, we propose DeepSTL, a method and associated tool for the translation of CPS requirements given in relatively free English to Signal Temporal Logic (STL) [2], a formal specification language used by the CPS academia and industry. To develop DeepSTL we address the following five research questions (**RQ**), the solutions of which are also our main contributions.

RQ1: What kind of empirical statistics of STL requirements, found in scientific literature, can guide data generation?

RQ2: How to generate synthetic examples of STL requirements consistently with the empirically collected statistics?

The first two research questions are related to **Challenge 1**. For **RQ1**, empirical STL statistics in literature and practice are analyzed in Section 4. For **RQ2**, we design in Section 5, a systematic grammar-based generation of synthetic data sets, consisting of pairs of STL formulae and their associated set of possible English translations.

RQ3: How effective is DeepSTL in learning synthetic STL?

RQ4: How well does DeepSTL extrapolate to STL requirements found in scientific literature?

RQ5: How do alternative deep learning mechanisms used in machine translation compare to DeepSTL's transformer architecture?

The last three research questions are relevant to **Challenge 2**. They are addressed and discussed in Section 6. We employ a corresponding transformer-based NLP architecture, whose attention

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

mechanism enables the efficient training of accurate translators from English to STL. We also compare DeepSTL with other machine translation techniques with respect to translating performance, on synthetic STL training and test data sets and their extrapolations.

2 RELATED WORK

From Natural Language to Temporal Logics During the last 25 years there has been a tremendous effort in developing techniques [3–16] for translating English requirements into temporal logics languages. Despite all these attempts, this problem is still considered an open challenge [9] due to the inherent difficulty of translating a sentence written in a natural and ambiguous language into a more general and concise formal language. All the available approaches rely on particular assumptions guiding the translation process. For example, they may require the user to express requirements in a restricted and controlled natural language [6, 12, 17] or to use some predefined specification patterns [4, 8, 18]. Other works [3, 11, 13, 15] enable the use of unconstrained natural languages, but they need first to translate them into intermediate representations and they require the manual specification of the rules/macros necessary to map the intermediate language into temporal logic patterns. These approaches mainly focus on the use of Linear Temporal Logic (LTL) [19], a temporal logic reasoning over logical-time Boolean signals.

In this paper we consider instead (for the first time to the best of our knowledge) the problem of automatic translation of unconstrained English sentences into Signal Temporal Logic (STL) [20], a temporal logic that extends LTL with operators expressing temporal properties over dense-time real-valued signals. STL is a well-established formal language employed in both academia and advanced industrial research labs to specify requirements for CPS engineering [21, 22].

Semantic Parsing Our problem is as an instance of semantic parsing, a task that consists in automatically mapping context-sensitive natural language sentences into utterances of a machine-executable language with a deterministic context free grammar. Notable frameworks to develop semantic parsing are SEMPRE [23], KRISP [24], SippyCup [25], WASP [26] and Cornell Semantic Parsing [27]. A typical application of semantic parsing is the automatic generation of Structured-Query-Language (SQL) queries [28–32] from questions formulated in natural language. Other applications include the translation of natural-language text into Python code [33], bash commands [34], and domain specific languages [35]. The main challenge for this task is how to obtain automatically suitable semantic derivation rules that can capture the full range of natural language contexts. Despite several approaches proposed to infer grammars and facilitating this task [36–38], they all require a large training data set of examples to be effective.

In order to cope with the lack of publicly available informal-requirement and formal-specification data sets, we first design a grammar-based generation technique of synthetic data, where each output is a random STL formula and its associated set of possible English translations. Then we address the problem as a neural machine translation, where a deep neural network is trained to predict, given the utterance in English, the likelihood of a STL formula expressing it. Our approach takes advantage of general-purpose ML frameworks such as PyTorch [39] or Tensorflow [40], and of state-of-the-art solutions based on transformers and their attention mechanisms [41].

3 SIGNAL TEMPORAL LOGIC (STL)

Signal Temporal Logic (STL) with both *past* and *future* operators is a formal specification formalism used by the academic researchers and practitioners to formalize temporal requirements of CPS behaviors. STL allows to express real time requirements of continuous-time real-valued behaviors. An example is a simple bounded stabilization property formulated as follows: *It is always the case that when the signal In is greater than 5, the signal Out becomes within 10 time units smaller than 2.*

The syntax of an STL formula φ over a set X of real-valued variables is defined by the grammar:

$$\varphi \quad := \quad x \sim u \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2 \mid \varphi_1 \mathbf{S}_I \varphi_2$$

where $x \in X$, $\sim \in \{<, \leq\}$, $u \in \mathbb{Q}$, $I \subseteq [0, \infty)$ is a non-empty interval. For intervals of the form $[a, a]$, we will use the notation $\{a\}$ instead.

With respect to a signal $w : X \times [0, d) \to \mathbb{R}$, the semantics of an STL formula is described via the satisfiability relation $(w, i) \models \varphi$, indicating that the signal w satisfies φ at the time index i:

$$\begin{array}{lcl}
(w,i) \models x \sim u & \leftrightarrow & w(x,i) \sim u \\
(w,i) \models \neg\varphi & \leftrightarrow & (w,i) \not\models \varphi \\
(w,i) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,i) \models \varphi_1 \text{ or } (w,i) \models \varphi_2 \\
(w,i) \models \varphi_1 \mathbf{U}_I \varphi_2 & \leftrightarrow & \exists j \in (i+I) \cap \mathbb{T} : (w,j) \models \varphi_2 \\
 & & \text{and } \forall i < k < j, (w,k) \models \varphi_1 \\
(w,i) \models \varphi_1 \mathbf{S}_I \varphi_2 & \leftrightarrow & \exists j \in (i-I) \cap \mathbb{T} : (w,j) \models \varphi_2 \\
 & & \text{and } \forall j < k < i, (w,k) \models \varphi_1
\end{array}$$

We use $\mathbf{S}$ and $\mathbf{U}$ as syntactic sugar for the *untimed* variants of the *since* $\mathbf{S}_{(0,\infty)}$ and *until* $\mathbf{U}_{(0,\infty)}$ operators. From the basic definition of STL, we can derive the following standard operators.

tautology	**true**	=	$p \vee \neg p$
contradiction	**false**	=	$\neg$**true**
disjunction	$\varphi_1 \wedge \varphi_2$	=	$\neg(\neg\varphi_1 \vee \neg\varphi_2)$
implication	$\varphi_1 \rightarrow \varphi_2$	=	$\neg\varphi_1 \vee \varphi_2$
eventually, finally	$\mathbf{F}_I\varphi$	=	**true** $\mathbf{U}_I\ \varphi$
always, globally	$\mathbf{G}_I\varphi$	=	$\neg\mathbf{F}_I\neg\varphi$
once	$\mathbf{O}_I\varphi$	=	**true** $\mathbf{S}_I\ \varphi$
historically	$\mathbf{H}_I\varphi$	=	$\neg\mathbf{O}_I\neg\varphi$
rising edge	**rise**(φ)	=	$\varphi \wedge \neg\varphi$ **S true**
falling edge	**fall**(φ)	=	$\neg\varphi \wedge \varphi$ **S true**

We can now formalize the rather verbose English description of the above *Bounded Response* requirement, with a succinct STL formula as follows: $\mathbf{G}(\text{In} > 5 \rightarrow \mathbf{F}_{[0,10]}(\text{Out} < 2))$.

This formula can be directly used during the verification of a CPS before it was deployed, or to generate a monitor, checking the safety of the CPS, after its deployment.

4 EMPIRICAL STL STATISTICS

In order to address the relative lack of publicly available STL specifications, we develop a synthetic-training-data generator, as described in Section 5. Instead of exploring completely random STL sentences, the generator should focus on the creation of commonly

used STL specifications. In addition, every STL formula shall be associated to a set of natural language formulations, with commonly used sentence structure and vocabulary.

We analyzed over 130 STL specifications and their associated English-language formulation, from scientific papers and industrial documents. The investigated literature covers multiple application domains: specification patterns [42], automatic driving [43–45], robotics [46–50], time-series analysis [51] and electronics [2, 52]. Although this literature contains data that is not statistically exhaustive, it still provides valuable information to guide the design of the data generator and address the research question **RQ1**.

We present our results on the statistical analysis of the STL specifications in Section 4.1 and of their associated natural-language requirements in Section 4.2.

4.1 Analysis of STL Specifications

We conducted two main types of analysis for the STL specifications encountered in the literature: (1) Identification of common temporal-logic templates, and (2) Computation of the frequency of individual operators. During analysis, we made several other relevant observations that we report at the end of this section.

4.1.1 STL-Templates Distribution. We identified four common STL templates that we call: *Invariance/Reachability*, *Immediate response*, *Temporal response* and *Stabilization/Recurrence*.

Invariance/Reachability template: Bounded and unbounded invariance and reachability are the simplest temporal STL properties. They have the form $\mathbf{G}A$, $\mathbf{G}_{[a,b]}A$, $\mathbf{F}A$ or $\mathbf{F}_{[a,b]}A$, where A is an atomic predicate. We provide one example of bounded-invariance (BI) [44], and one example of unbounded-reachability (UR) [47] specification, respectively, as encountered in our investigation:

$$BI : \mathbf{G}_{[\tau_s,T]}(\mu < c_l), \quad UR : \mathbf{F}(x > 0.4)$$

Immediate response template: This template represents formulas of the form $\mathbf{G}(\mathrm{A} \rightarrow \mathrm{B})$, where A and B are atomic propositions or their Boolean combinations. Except for the starting **G** operator, there are no other temporal operators in the formula. An example of an immediate response (IR) specification is the one from [42]:

$$IR : \mathbf{G}(\text{not\_Eclipse} = 0 \rightarrow \text{sun\_currents} = 0).$$

Temporal response template: This template represents formulas of the form $\mathbf{G}(\varphi \rightarrow \psi)$, where φ and ψ can have non-nested temporal operators. We illustrate several TR specifications that we encountered in the literature. They all belong to this class:

$$TR1 : \mathbf{G}(\mathbf{rise}(\text{Op\_Cmd} = \text{Passive}) \rightarrow \mathbf{F}_{[0,500]}\text{Spd\_Act} = 0) \quad [53]$$
$$TR2 : \mathbf{G}(\text{currentADCSMode} = \text{SM} \rightarrow P\,\mathbf{U}_{[0,10799]}\,\neg P \quad [42]$$
$$TR3 : \mathbf{G}(\mathbf{rise}(\text{gear\_id} = 1) \rightarrow \mathbf{G}_{[0,2.5]}\neg\mathbf{fall}(\text{gear\_id} = 1)) \quad [43]$$

In $TR2$ above, $P \equiv \text{real\_Omega} - \text{target\_Omega} = 0$.

Stabilization/Recurrence template: These templates represent formulas allowing one nesting of the temporal operators. Typical nesting is $\mathbf{GF}\,\varphi$ for recurrence (RE), and $\mathbf{FG}\,\varphi$ for stabilization (ST), with their bounded counterparts. Here φ is a non-temporal formula. The following specifications from the literature are in this category:

$$ST : \mathbf{F}_{[0,14400]}\mathbf{G}_{[4590,9963]}\ (x_{10} \geq 0.325) \quad [51]$$
$$RE : \mathbf{G}_{[0,12]}(\mathbf{F}_{[0,2]}\ \text{regionA} \wedge \mathbf{F}_{[0,2]}\ \text{regionB}) \quad [47]$$

Other formulas: These are formulas that do not fall into any of the above categories. The following specification belongs to this class:

$$\mathbf{G}(\mathbf{rise}(\varphi_1) \rightarrow \mathbf{F}_{[0,t_1]}(\mathbf{rise}(\varphi_2) \wedge (\varphi_2\,\mathbf{U}_{[t_2,t_3]}\,\varphi_3)))$$

It captures the following requirement: *Whenever the precondition φ_1 becomes true, there is a time within t_1 units where φ_2 becomes true and continuously holds until φ_3 becomes true within the interval $[t_2, t_3]$*. This pattern is used in the electronics field [2] to describe the situation where one digital signal tracks another [52].

Statistics: We encountered 39 Invariance/Reachability (30.0%), 27 Immediate Response (20.8%), 33 Temporal Response (25.4%) and 31 Stabilization/Recurrence (23.8%) templates. The category of Other templates is orthogonal to the first four ones since it includes ad hoc formulas. There are overall 13 (33.4%) Temporal Response and 6 (19.3%) Stabilization/Recurrence templates belonging to this type.

4.1.2 STL-Operators Distribution. We investigated the distribution of the STL-operators as encountered in the specifications found in the above-mentioned literature. Figure 1 summarizes our results.

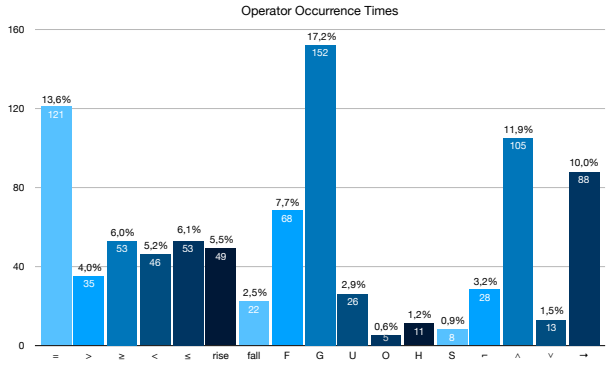

Figure 1: Frequency Distribution of STL Operators.

We first discuss the (atomic) numeric predicates. We observe that the equality operator occurs more often in numerical predicates then the other ($\leq$, $<$, $\geq$, $>$) relations. This happens because many specifications refer to discrete mode signals and equality is used to check if a discrete variable is in a given mode.

Conjunction and implication are the two most frequently used Boolean operators. Conjunction is often used to specify that a signal must lie within a given range. Implication is used in a pretty wide range of response specifications.

The **rise** and **fall** operators are typically used in front of an atomic predicate (for example $\mathbf{rise}(x > \mu)$). The frequency of rising edges is higher than that of falling edges, which can be explained by the fact that many specifications refer to time instants where a condition starts holding, rather than when it stops holding.

The **G** operator has a much higher frequency than any other temporal operator. This is not surprising because 87.7% (114/130) of the specifications are invariance or response properties that start with an always operator. The **F** operator ranks second and is often used in specifications of robotic applications to define reachability objectives. We also remark that "eventually" is used in bounded/unbounded and stabilization properties.

We finally observe that future temporal operators (**G**, **F**, **U**) are used more often than their past counterparts (**O**, **H**, **S**) and that unary temporal operators (**G**, **F**, **H**, **O**) are used more often than the binary ones (**U**, **S**). These two observations are explained by the fact that most declarative specifications have a natural future flavor (a trigger now implies an obligation that must be fulfilled later) and unary temporal operators are easier to understand and handle.

4.1.3 Other Observations. In this section, we discuss additional findings that we discovered during the analysis of the STL specifications occurring in the literature:

- We found a frequent usage of the pattern $|x - y|$ to denote the pointwise distance between signals x and y, especially in the motion control applications [45–47].
- Some publications use abstract predicates to denote complex temporal patterns, without providing their detailed formalization. One such example is the use of the predicate spike(x) to denote a spike occurring within the signal x [42].
- It is relatively common in the literature to decompose a complex STL specification into multiple simpler ones, by giving a name to a sub-formula and using that name as an atomic proposition in the main formula.
- Time bounds in temporal operators and signal thresholds are sometimes given as parameters, rather than constants.
- **rise**, **fall** and past temporal operators are normally used as pre-conditions, while future operators are often used as post-conditions. Negation is used conservatively, e.g., ¬**fall** is used to represent a particular stabilized condition should hold for a designated time interval [2].

4.2 Analysis of NL Specifications

In this section we investigate the usage of natural language (NL) in the literature to express informal requirements, which are then formalized using STL. In particular, we identified the English vocabulary used to formulate STL operators and sentences, and studied the quality, accuracy and preciseness of the language.

4.2.1 English formulation of STL sentences. We considered several aspects (e.g., nouns, verbs, adverbs, etc.) when studying the use of natural language in the formulation of:

- Numeric (atomic) predicates,
- Temporal operators (phrases),
- Specific scenarios (e.g., a rising/falling edge).

The main outcome of this analysis is that the language features used in the studied requirements are unbalanced and sparse and that it is hard to identify a general recurring pattern. We illustrate this observation with two representative examples:

- **Example 1**: We counted different English utterances to express the semantics of $x > \mu$. The most frequently used collocation is "be above", which appears overall 4 times. Next comes "increase above" (somewhat ambiguous because this may also represent rising edge), which is used 2 times. Then "be higher than", "be larger than" and "be greater than" are only used once respectively. However, we do not find any requirements using other synonymous expressions like "be more than" or "be over".
- **Example 2**: We observed that two temporal adverbs are frequently used to express $\mathbf{G}_{[0,t]}$ and $\mathbf{H}_{[0,t]}$, which are "for at least t time units (s, ms, etc.)" (9 times) and "for more than t time units" (6 times). However, other reasonable possibilities like "for the following/past t time units" are not found.

The sparsity and lack of balance may be a consequence of the relative small base of publicly available literature that defines this type of requirements. Despite the fact that the findings of this analysis may not be sufficiently representative, we can still use the outcomes to improve our synthetic generation of examples.

4.2.2 Language Quality. For the English requirements found in the literature, of particular interest is the language quality: How accurately does a requirement reflect the semantics of its corresponding STL formula? Given this criterion, we classify the studied English requirements into *clear*, *indirect* and *ambiguous* requirements.

Clear: These requirements have a straightforward STL formalisation that results in an unambiguous specification without room for interpretation. An example of a clear requirement is the sentence: *If the value of signal control_error is less than* $10°$*, then the value of signal currentAD-CSMode shall be equal to NMF* [42]. The resulting STL specification is given by the formula:

$$\mathbf{G}(\text{control\_error} < 10 \rightarrow \text{currentAD-CSMode} = \text{NMF})$$

Indirect: These requirements need an expert to translate them into an STL formula that faithfully captures the intended meaning. They typically assume some implicit knowledge that must be added to the formal specification from the context. An example is the sentence: *The vehicle shall stay within the lane boundaries, if this is possible with the actuators it is equipped with* [45]. This is an indirect requirement formalized using the following STL formula:

$$\mathbf{G}(\tau < \tau_{max} \rightarrow \mathbf{P})$$

Here **P** is the contextual sub-formula: *vehicle* $\subseteq$ *corridor*.

Ambiguous: These requirement lack key information that cannot be easily inferred from the context and that must be extracted from external sources, such as tables, figures, timing diagrams, or experts. They use vague and ambiguous language, and can have multiple interpretations. An example is the following sentence: *To prevent the destruction of the device by avalanche due to high voltages, there is a voltage clamp mechanism* $Z_{DS(AZ)}$ *implemented, that limits negative output voltage to a certain level* $V_S - V_{DS(AZ)}$*. Please refer to Figure 10 and Figure 11 for details* [52]. This is an ambiguous requirement that can be translated to the following STL formulas:

$$\mathbf{G}(V_{OUT} < V_{GND} \wedge I_L > 0 \rightarrow V_{OUT} = V_S - V_{DS(AZ)})$$
$$\mathbf{G}(V_{OUT} < V_{GND} \rightarrow V_{OUT} = V_S - V_{DS(AZ)})$$

The English requirement only vaguely mentions the post-condition. The pre-condition characterizes the drop of voltage V_{OUT} below V_{GND} when the inductive load is being switched off. This is obtained from the previous context and Figure 11 of [52] with some physical knowledge that inductive current has to change smoothly.

We encountered 46 Clear (35.4%), 43 Indirect (33.1%), and 41 Ambiguous (31.5%) English requirements.

5 CORPUS CONSTRUCTION

This section addresses research question **RQ2**. It first introduces a new method for the automatic generation of STL sentences and their associated natural language requirements. The generator incorporates the outcomes from Section 4 for improved results. Finally, we use this method to do the actual generation of STL-specification/NL-requirements pairs.

5.1 Corpus Generation

In the following, we propose an automatic procedure for randomly generating synthetic examples. Each example consists of: (1) An STL formula, and (2) A set of associated sentences in English that describe this formula. We associate multiple natural-language sentences to each formal STL requirement to reflect the fact that formal specifications admit multiple natural language formulations. We illustrate this observation using the *bounded response* specification from Section 3, formalized as the STL formula below:

$$\mathbf{G}(\text{In} > 5 \rightarrow \mathbf{F}_{[0,10]}\text{Out} < 2))$$

This admits multiple synonymous English formulations, including:

- Globally, if the value of In is greater than 5, then finally the value of Out should be smaller than 2 at a time point within 10 time units.
- Whenever In is above 5, then there must exist a time point in the next 10 time units, at which the value of Out should be no more than 2.

This example shows that two NL formulations of the same STL formula can be very different, making the generation of synthetic examples a challenging task. The systematic translation of unrestricted STL is indeed extremely difficult, especially for specifications that include multiple nesting of temporal operators. In practice, deep nesting of temporal formulas is rarely used because the resulting specifications tend to be difficult to understand.

Hence, we first restrict STL to a rich but well-structured subfragment that facilitates a fully automated translation, while at the same time covering commonly used specifications.

5.1.1 Restricted-STL Fragment. In this subsection, we present the restricted fragment of STL that we support in our synthetic example generator. We define this fragment using three layers that can be mapped to the syntax hierarchy identified in Section 4.1.1.

The bottom layer, which we call **simple-phrase (SP)** layer, consists of: (1) **Atomic propositions (α)** including rising and falling edges and (2) **Boolean combinations** of up to two atomic propositions.

$$\begin{aligned} \alpha &:= x \circ u \mid \neg(x \circ u) \mid \mathbf{rise}(x \circ u) \mid \mathbf{fall}(x \circ u) \mid \\ &\quad \neg\mathbf{rise}(x \circ u) \mid \neg\mathbf{fall}(x \circ u) \\ \text{SP} &:= \alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \end{aligned}$$

where x is a signal name, u a constant, and $\circ \in \{<, \leq, =, \geq, >\}$.

The middle layer, which we call **temporal-phrase (TP)** layer, admits the specification of temporal formulas over simple phrases:

$$\begin{aligned} \text{TP} &:= \text{TP}' \mid \neg\text{TP}' \mid \mathbf{rise}\ \text{TP}' \mid \mathbf{fall}\ \text{TP}' \mid \neg\mathbf{rise}\ \text{TP}' \mid \neg\mathbf{fall}\ \text{TP}' \\ \text{TP}' &:= \mathbf{UTO}_I\ (\alpha) \mid (\alpha)\ \mathbf{BTO}_I\ (\alpha) \end{aligned}$$

where $\mathbf{UTO} \in \{\mathbf{F}, \mathbf{G}, \mathbf{O}, \mathbf{H}\}$ and $\mathbf{BTO} \in \{\mathbf{U}, \mathbf{S}\}$ are unary and binary temporal operators, respectively. I is an interval of the form $[t_1, t_2]$ with $0 \leq t_1 < t_2 \leq \infty$. This can be omitted if $t_1 = 0$ and $t_2 = \infty$.

The top layer, which we call single **nested-temporal-phrase (NTP)** layer, allows the formulation of formulas with a single nesting of a subset of temporal operators:

$$\text{NTP} := \mathbf{F}_I\mathbf{G}_I(\alpha) \mid \mathbf{G}_I\mathbf{F}_I(\alpha)$$

where I follows the same definition as mentioned above.

Finally, with an auxiliary syntactical component P := SP | TP, formula ψ defines the **supported fragment** of STL that we map to the four template categories discussed in Section 4.1.1.

$$\begin{aligned} \psi &:= \mathbf{G}_I(\text{SP}) \mid \mathbf{F}_I(\text{SP}) && (\textit{Invariance/Reachability}) \\ &\mid \mathbf{G}\ (\text{SP} \rightarrow \text{SP}) && (\textit{Immediate response}) \\ &\mid \mathbf{G}\ (\text{P} \rightarrow \text{TP}) && (\textit{Temporal response}) \\ &\mid \mathbf{G}\ (\text{P} \rightarrow \text{NTP}) && (\textit{Stabilization/Recurrence}) \end{aligned}$$

This fragment balances between *generality*, needed to express common-practice requirements, and *simplicity*, needed to facilitate the automated generation of synthetic examples. It results in the following restrictions: (1) We allow the conjunction and disjunction of only two atomic propositions, (2) Only one atomic proposition is allowed inside a temporal operator in **TP**, (3) We do not allow Boolean combinations of **SP** and **TP** formulas, and (4) Formulas outside the four mentioned templates are not supported.

By relating the generator-fragment ψ to the empirical statistics in Section 4.1.1, Figure 2 summarizes for each syntactical category, the proportion of templates that the fragment can support.

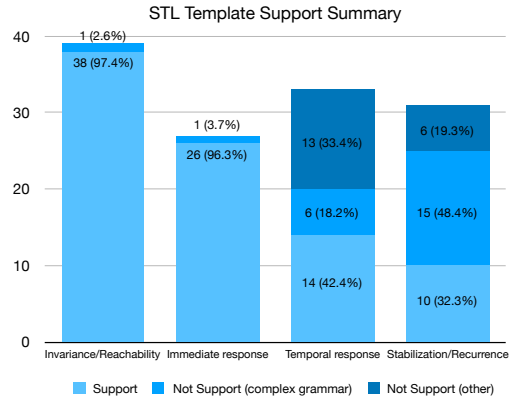

Figure 2: STL Template Support Summary.

The generator nearly supports all *Invariance/Reachability* templates appearing in our database. For *Immediate Response* ones, there is one template missing due to restriction (1). For *Temporal Response* templates, we are able to support 42.4% of them. For the not supported ones, 18.2% use a complex grammar that violates restriction (2) and (3), while the remaining ones (33.4%) belong to the *other category* for ad hoc purposes. Concerning nesting formulas, we only consider stabilization and recurrence templates. Other combinations such as $\mathbf{FF}\varphi$ or $\mathbf{F}\varphi_1\mathbf{U}\varphi_2$ are not supported: 48.4% of them are in the *complex grammar* group, while the other 19.3% are in the *other category*.

5.1.2 Random-Sampling STL Formulas. This short subsection briefly describes how we sample STL specifications from the restricted fragment. The main idea is to decorate the grammar rules with probabilities according to the template distribution collected in Section 4.1 and the operator distribution shown in Figure 1.

Consequently, we use the probabilities described in Section 4.1 to generate the four categories of fragment ψ, which will naturally make the **G** operator rank first to a large extent, followed by the **F** operator, regarding to usage frequency. The frequencies of the other operators within these categories are as discussed above.

5.1.3 Translating STL into English. The main translation strategy linked to 4.2 is as follows. For the predicates used to express logical relations in the bottom layer, we use (with some reservations mainly with regards to accuracy) the frequencies of Section 4.2.1. This way, the translation candidates are selected with different weights. For the others, such as the adverbs specifying temporal information, we incorporated relevant English utterances encountered in our database, on the condition that the generation and recognition can be done with both accuracy and fluency. Besides, much room has been reserved to add synonymous utterances that have not appeared in the database but conform to common usage habits.

In order to systematically organize the translation and maximize language flexibility, we start with the translation of atomic propositions (defined as α in 5.1.1) in the bottom syntactical layer, and use this as a pivot to tackle temporal phrases and their nesting scenarios in the middle and top layers.

Bottom layer. The English counterpart of atomic propositions typically consist of a subject, a predicate, and an object. They are indispensable in each English sentence. Hence, their variations especially in the predicate (including the choice of verbs, formats, tenses, and their active/passive voice) are considered first. The workflow for the organisation of their translations, is divided into a *Handler* and a *Translator*, is illustrated in Figure 3.

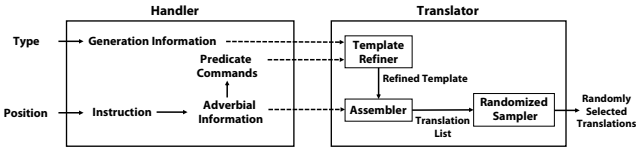

Figure 3: Translation Procedure for Atomic Propositions.

The Handler, as a preprocessor, takes the *type* and *position* information as inputs. Type is a branch of α used to compute and output the *Generation-Information*. This includes an *index* (trigger corresponding to a translation strategy), *identifiers*, *numbers*, and the *STL expression* of a randomly generated atomic proposition.

Position specifies the location of the proposition. This determines if the translation states a certain scenario, if it is a condition (before implication symbol "→"), or if it emphasises that a property has to hold with a satisfied condition (after "→"). In the latter case, modal verbs like "should" or"must" need to be used, and often together with adverbial modifiers like "instantly" or "without any delay" in case of *Immediate response* formulas. This information is embedded into *Predicate-Commands*, incorporating the choice of verbs, their format, and the use of modal verbs and of adverbial modifiers.

Generation-Information and Predicate-Commands are sent to the *Template-Refiner* (inside Translator), whose architecture is shown in Figure 4. Here, the subject and object placeholders within the templates are replaced by randomly generated identifiers and numbers. The verbs associated to predicates are changed to their proper format, and are decorated with adverbs when applicable.

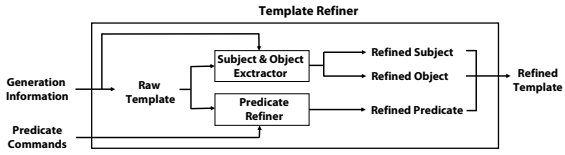

Figure 4: Template Refiner.

In the next step, the *Assembler* module of the Translator completes the refined templates into a complete sentence that also includes adverbial modifiers. Finally, the *Randomized-Sampler* module of the Translator, samples a designated number of sentences from the overall translation list.

Middle/Top layer. The translation approach presented above is extended to temporal phrases in a straight-forward manner, because the sentences generated by the bottom layer, can be reused except for the need to add adverbial modifiers, and enrich the verb tenses according to the temporal operators.

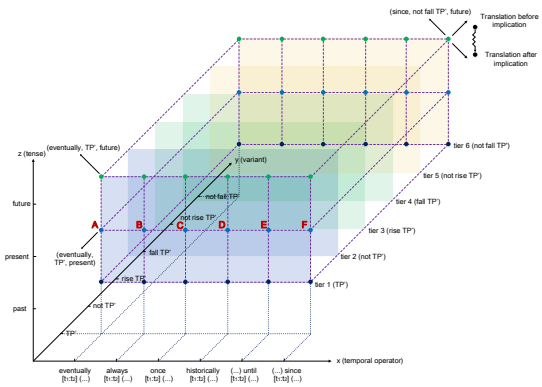

Figure 5: Translation of Temporal Phrases.

The temporal aspects nevertheless do increase the translation complexity. We need to consider three orthogonal aspects (dimensions) as shown in Figure 5. The x-axis represents the six STL temporal operators from the *TP* layer, the y-axis their variants preceded by the negation, rising or falling edge operators, and the z-axis the choice of a verb tense in English for specific temporal operators. Hence, a node in Figure 5 represents a specific combination of these three aspects.

We adopt a slicing approach to tackle the complexity. We first process nodes A-F with present tense, where the six temporal operators are used individually. Then we enrich the usage of verb tenses according to the semantics of a particular operator and its nesting situation. This results in tier TP′ while preserving language flexibility. The same approach is used for processing unary operators

in tier ¬TP′. The semantics of direct negation of binary operators, rising/falling edges and their negations for layer TP are complicated. Considering their relatively low usage frequency, we provide several fixed templates to facilitate their translations.

5.2 Corpus Statistics

Following the approach described in Section 5.1, we have automatically generated a corpus consisting of 120,000 formula-text pairs where each pair consists of a randomly generated STL formula and one of its generated translation in natural language.

5.2.1 STL-Formula Statistics. In Figure 6 we provide the frequencies of the STL operators in our synthetic dataset (above corpus). As one can see they are consistent with the ones in Figure 1. As before, the most frequent STL operator is global temporal operator $\mathbf{G}_I\varphi$ with more than 138,715 occurrences. The least frequent STL operator is the $\varphi_1\mathbf{S}_I\varphi_2$ temporal operator with approximately 5,105 occurrences. While this frequency differs a bit from Figure 1, it is still consistent with the empirical results.

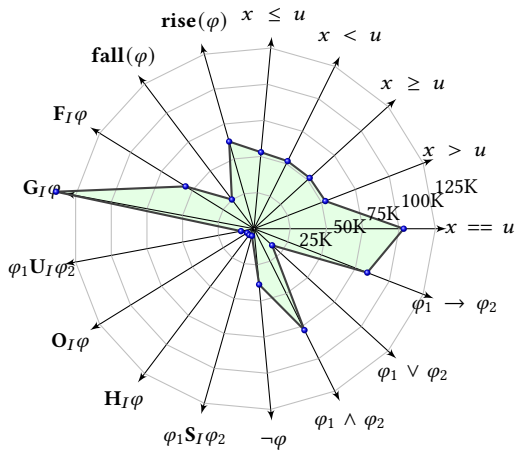

Figure 6: Frequency of STL operators in the corpus.

Table 1 shows the statistics of templates and subformulas in the generated corpus. As mentioned in Section 4.1, an STL template is defined as the parse tree of a formula without its leaves. For example, the template for the formula $\varphi = \mathbf{G}(\text{In} > 5 \rightarrow \mathbf{F}_{[0,10]}\ \text{Out} < 2)$ is $\mathbf{G}(\varphi_1 \rightarrow \mathbf{F}_{[0,10]}\varphi_2)$. Each formula has a finite number of subformulas. For example the formula φ above has five subformulas: $\varphi_5 = \mathbf{G}(\text{In} > 5 \rightarrow \mathbf{F}_{[0,10]}\ \text{Out} < 2)$, $\varphi_4 = \text{In} > 5 \rightarrow \mathbf{F}_{[0,10]}\ \text{Out} < 2$, $\varphi_3 = \mathbf{F}_{[0,10]}\text{Out} < 2$, $\varphi_2 = \text{Out} < 2$, and $\varphi_1 = \text{In} > 5$.

Table 1: STL Formula Statistics: # unique STL formulas, # unique STL templates, # subformulas for each formula.

# formulas	# templates	# subformula per formula			
		min	max	avg.	median
120,000	5,852	2	18	6.98	7

Table 2 shows the mutual mapping relation between STL operators and STL formulas in our corpus. We count for each STL operator, how many formulas it has appeared in. This produces the containment statistics shown in the last three columns.

Table 2: STL-Formula Mapping Statistics: # STL operators for each formula, # STL formulas for each operator.

# STL oper. per formula			# formulas per STL oper.		
avg.	median	max	avg.	median	max
6.98	7	18	42,120.3	45,125	101,870

Since identifiers and constants frequently appear in our corpus, we also analyzed their frequency, as shown in Table 3.

Table 3: Identifier and Constants Statistics: average number of identifiers per formula, # of chars used per identifier, # number of digits used per constant.

# identifiers	# chars per identifier			# digits per constant		
per formula	min	avg.	median	min	avg.	median
2.59	1	5.50	5	1	2.31	2

5.2.2 Natural-Language Statistics. The statistical results of the natural language in our corpus are shown in Table 4. There are only 265 different effective English words (considering word variants, not including signal names which are strings generated randomly), constituting a relatively small vocabulary. This is understandable because most English words are used to express the logical relation in STL, the number of which is thus limited. Table 4 also records the statistics of effective word numbers in all English sentences. It counts for each English word, the number of English sentences using it.

Table 4: English Statistics: # unique sentences, # unique words, # words per sentences and # sentences per word.

# sent.	# word	# words per sent.		# sent. per word	
		avg.	median	avg.	median
120,000	265	38.49	37	14,220.28	4,555.5

6 MACHINE TRANSLATION

In order to answer questions **RQ3-5**, we take advantage of the corpus generated as discussed in the previous sections, to develop DeepSTL, a tool and technique for the translation of informal requirements given as free English sentences, into STL. DeepSTL employs a state-of-the-art transformer-based neural-translation technique, to train an accurate attentional translator. We compare the performance of DeepSTL with other NL translator architectures on the same corpus, and we also investigate how they are able to extrapolate to sentences out of the corpus.

6.1 Neural Translation Algorithms

The translation of natural language into STL formulas can be abstracted as the following probabilistic problem. Given an encoding sequence $\mathbf{e} = (e_1, e_2, ..., e_m)$ from the source language (English requirements), a decoding sequence $\mathbf{s} = (s_1, s_2, ..., e_n)$ from the target language (STL formulas) generates all of its tokens s_k conditioning on the decoded history of the target sequence $s_{<k}$ and

the whole input of the source sequence $\mathbf{e}$ such that: $P(\mathbf{s}|\mathbf{e};\theta) = \prod_{k=1}^{n} P(s_k|s_{<k}, \mathbf{e};\theta)$ where θ are the parameters of the model. A current practice in the community of NLP is to learn these probabilities through Neural Translation (NT) where the tokens are encoded into real vectors.

6.1.1 NT-Architectures considered. We considered three main NT-architectures: sequence to sequence (seq2seq), sequence to sequence plus attention (Att-seq2seq), and the transformer architecture.

Seq2seq architecture. Seq2seq uses two recurrent neural networks (RNNs), one in the encoder, and one in the decoder, to sequentially process the sentences, word by word [54].

Att-seq2seq architecture A drawback of the seq2seq architecture, is that it gradually encodes the dependencies among words in the input and output sentences, by sequentially passing the information to the next cell of the RNN. As a consequence, far-away dependencies may get diluted. In order to correct this problem, an attention mechanism is introduced in att-seq2seq, to explicitly capture and learn these dependencies [55].

Transformer-architecture The previous two architectures are relatively slow to train, because the RNNs hinder parallel processing. To alleviate this problem, the transformer architecture, introduces a self-attention mechanism, dropping completely the use of RNNs. This dramatically speeded up the computation time of attention-based neural networks, and conferred a considerable momentum to NT [41]. For this reason we adopted a transformer-based architecture for our DeepSTL translator.

6.1.2 Preprocessing and Tokenization. There are three main features that distinguish our translation problem from general translation tasks between natural languages (NL2NL):

(1) *Out of Vocabulary (OOV)*: The signal names, we call them identifiers for short, and numbers, can be arbitrarily specified. Therefore, it is impossible to maintain a fixed-size vocabulary to cover all imaginable identifiers and numbers.
(2) *High Copying Frequency:* During translation, identifiers and numbers need to be much more frequently copied from the source language to the target language than in NL2NL.
(3) *Unbalanced Language:* English is a kind of high-resource language while STL formulas belong to a low-resource logical language that has very limited exclusive vocabulary.

In view of the above characteristics, a successful translation of English to STL requires more than in NL2NL, a correct tokenization of identifiers and numbers. Although one can use an explicit copying mechanism [56], this method requires to modify the structure of the neural network, which increases complexity.

Subword tokenization We therefore adopt a subword technique to tokenize sequences during data preprocessing. Subword algorithms, such as Byte-Pair-Encoding (BPE) [57], WordPiece [58] and Unigram [59], are commonly used in state-of-art NT systems to tackle the OOV problem. Without modifying the model structure, these algorithms are able to split words into meaningful morphemes and even independent characters based on statistical rules.

Ideally, we hope that when tokenizing identifiers and numbers, they can be respectively split into characters and digits separated by whitespaces. For example, assuming ' ' represents one whitespace, then "In" and "12.5" are expected to get encoded as ['I ', 'n '] and ['1 ', '2 ', '. ', '5 '], respectively. This way, we can use a limited number of characters and digits to represent arbitrary identifiers and numbers. We chose the BPE, because of its simplicity.

BPE is executed as follows: (1) Split every word (separated by space) in the source data to a sequence of characters. (2) A prepared token list will include all possible characters (without repetition) in the source data. (3) The most frequently occurring pair of characters are merged and added to the token list, then this pair will be treated as an independent character afterwards. (4) Step 3 is repeated until the size of the token list reaches to an upper limit or a specified hyperparameter. When encoding a sequence, the generated list is iterated from the longest token to the shortest token attempting to match and substitute substrings for each word in the sequence.

Inspired by above BPE execution, we split identifiers and numbers into characters and digits separated by a whitespace in the pre-tokenization phase, such that the characters and digits will not participate in the merging procedure of BPE. Besides, this operation will also benefit encoding, during which only characters and digits in the generated token list can match identifiers and numbers since they are already split into separate units.

During testing time, although it is easy to use regular expressions to match numbers and split them into digits, it is challenging to accurately match identifiers. This is because identifiers can be non-meaningful permutations of characters, or complete English words. These two scenarios cannot be easily distinguished. An ideal method is to adopt Name Entity Recognition (NER) to match identifiers and split them. We leave this for future work. For now, we only manually separate identifiers to verify the feasibility of using subword techniques to solve our specific translation problems.

6.2 Implementation Details

6.2.1 Data split. We overall generated 120000 English-STL pairs, from which we first sampled 10% (12000) to prepare a fixed testing set. For the rest, before each training experiment, we sampled 90% (97200) of them for training, and 10% (10800) for validation.

6.2.2 Hyperparameters. The implementation of the three models mentioned in 6.1.1 are mainly based on [60] with several modifications using Pytorch. The following describes how hyperparameters are chosen for each model and the optimizer.

Seq2seq We used (Gated Recurrent Unit, GRU) [61] as RNN units. The encoder is a 2-layer bidirectional RNN, and the decoder is a 2-layer unidirectional RNN. For each GRU unit, hidden size $h = 128$. The embedding dimension for mapping a one-hot vector (represents a token) into real valued space is 128. Drop out rate is 0.1.

Att-seq2seq For the encoder-decoder, we used the same hyperparameters as Seq2seq-architecture. For Bahdanau attention [55], we used a 1- hidden layer feed-forward neural network with 128 neurons to calculate attention score.

Transformer For the encoder and the decoder, they both have 4 layers with 8 attention heads; Input and Output dimensions for each computing block are always preserved as $d_{\text{model}} = 128$; Neuron number in feed-forward layers equals to $d_{ff} = 512$; Drop out rate is 0.1; Layer normalize epsilon is 10^{-5}.

Optimizer We used Adam Optimization algorithm [62] with $\beta_1 = 0.9$, $\beta_1 = 0.98$, $\epsilon = 10^{-9}$, while the learning rate lr is dynamically

scheduled as (slightly changed from [41]):

$$lr = p \cdot d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num^{-0.5} \cdot warmup_{steps}^{-1.5})$$

where $warmup_{steps} = 4000$, $d_{\text{model}} = 128$. $step\_num$ represents the training steps (training on one batch corresponds to one step). p is an adjustable parameter for each architecture, and we chose 1, 0.1 and 2 for Seq2seq, Att-seq2seq and Transformer respectively. For Seq2seq and Att-seq2seq models, in order to ease gradient explosion due to long sequence dependency, we also used gradient clipping to limit the maximum norm of gradients to 1.

Other We dealt with variable length of input and output sequence using padding. We firstly encoded all English and STL sequences into subword token lists, from which we picked the maximum length as the step limit both for the encoder and the decoder. During training, for sequence whose length is smaller than the maximum value, we padded a special token '<pad>' to its end for complement.

6.2.3 Train/Validate/Test Procedure. For training and validation, we used "teacher forcing" strategy in the decoder. We firstly prepared two special tokens '<bos>' (begin of sentence) and '<eos>' (end of sentence). Suppose the reference output of the decoder is "ABC<eos>". To start with, we input '<bos>' as a starting signal to the decoder and hoped that it could output "A". No matter whether the actual first output of the decoder is "A", we then sent "A" to the decoder, and hoped that it would output "B". This procedure continues until the maximum step length of the decoder is reached.

We then summed up all token-level (only for valid length) cross-entropy loss between the prediction and reference sequence, divided by the maximum length of the decoder. This is the loss calculated for one sample. We trained a batch of 64 samples in parallel. The batch loss is averaged over all sample losses inside the batch, which will be used for back propagation to update network parameters.

For testing, "teacher forcing" is abandoned. The only token manually input to the decoder is '<bos>' for initialization. At each time step of decoding, the decoder adopts a greedy search strategy, outputting a token with maximum probability only based on its output in the previous step and the output of the encoder. The decoding procedure will end until the decoder outputs an '<eos>' token or the maximum limit length is reached.

6.3 Results

6.3.1 Loss/Accuracy Curves. We trained Seq2seq, Att-seq2seq and Transformer architectures for 80, 10 and 40 epochs respectively, using *STL formula accuracy* (defined in 6.3.2) in validation as an indicator to stop training. The validation loss/accuracy curves are obtained by 5 independent experiments and shown as follows.

Figure 7 and Figure 8 show that, with the guidance of "teacher forcing", all the three models are able to converge during training, making the STL formula accuracy approach to 1 when the network becomes stabilized. The only difference is the rate of convergence, which depends on many factors like the volume of the model (e.g., number of parameters), noises, learning rate, etc.

6.3.2 Testing Metrics. We firstly report two different measures of accuracy: the *STL formula accuracy* (A_F) and the *template accuracy* (A_T). The first measures the alignment accuracy for the reference and prediction sequence in a string level, while the second firstly

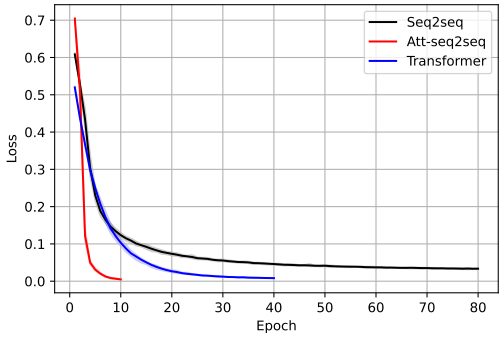

Figure 7: Validation Loss.

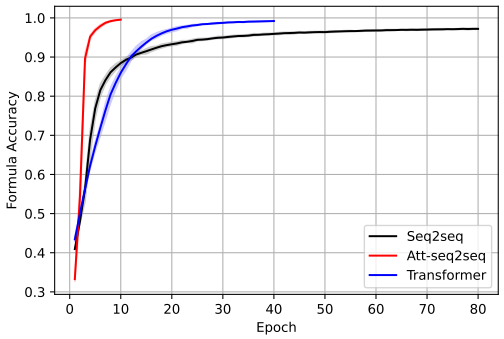

Figure 8: Validation Formula Accuracy.

transforms the reference and predication instances into STL templates and then calculates their alignment accuracy. For example,

Formula: always (x > 0) ⇒ Template: always (phi)
Formula: always (y > 0) ⇒ Template: always (phi)

The first line is reference sequence and the second line represents model prediction. For better illustration, we insert a white-space between each token and hence there are six tokens for formulas, and four tokens for templates. For formulas, there are five tokens appearing in the same position ('always', '(', '>', '0', ')'), while the left one token 'x' in the reference is mistranslated to 'y'. Therefore, the formula accuracy $A_F = 5/6$. As for the template, since all tokens are aligned with each other, the template accuracy $A_T = 1$.

We also report another metric called BLEU (Bilingual Evaluation Understudy) [63] that has been pervasively used in machine translation research. It evaluates the number of n-grams appearing in the reference sequence. The best BLEU score for a pair of sequences is 1, which means complete overlapping.

Table 5: Testing Accuracy.

	Formula Acc.	Template Acc.	BLEU
Seq2Seq	0.071 ± 0.0388	0.207 ± 0.0868	0.092 ± 0.0361
Att-seq2Seq	0.977 ± 0.0060	0.980 ± 0.0063	0.996 ± 0.0011
Transformer	0.987 ± 0.0028	0.995 ± 0.0014	0.998 ± 0.0005

In Table 5, it can be seen that once "teacher forcing" is removed, the performance of Seq2seq architecture decreases dramatically, which is partly due to its lack of attention mechanism to realize self-correction. For the other two models, both of them can achieve very high accuracy, with Transformer slightly better than Att-seq2seq. Since the testing data and training data are sampled from the same

data-set, in this sense, these two models show high translation quality when the distribution of language patterns in testing cases are similar to the training data. We also find that the template accuracy is higher than the formula accuracy. This phenomenon is understandable - once one formula is transformed into the form of template, the potential translation errors in identifiers, constants and logical relation symbols are masked.

6.3.3 Extrapolation. In the following, we use the informal requirements that we identified from the literature in Section 4 to evaluate how well the machine learning algorithm generalizes the translation outside of the training and validation data set.

In order to have a fair evaluation, we used the 14 **Clear** requirements (10% of the entire set) with the template structure supported by our tool. We pre-processed the requirements to remove units that are not supported by our tool. Table 6 summarizes the accuracy results for the three learning approaches. We see that with non-synthetic examples the formula accuracy drops considerably for all algorithms, while the average template accuracy remains relatively high (89.9%) for the Transformer approach. We believe that higher availability of publicly available informal requirements that could be used for training would considerably help improving the accuracy of the approach.

Table 6: Extrapolation Accuracy.

	Formula Acc.	Template Acc.	BLEU
Seq2Seq	0.050 ± 0.0283	0.158 ± 0.0895	0.027 ± 0.0120
Att-seq2Seq	0.559 ± 0.0865	0.742 ± 0.0660	0.888 ± 0.0348
Transformer	0.712 ± 0.0678	0.899 ± 0.0100	0.962 ± 0.0030

In the following, we provide three examples that illustrate the possibilities and the limits of our approach (random seed = 100). The part that is not correctly translated is represented in blue colour.

Example 1: If the value of signal RWs_angular_momentum is greater than 0.35, then the value of signal RWs_torque shall be equal to 0. [42]

- **Transformer**:
  always (RWs_angular_momentum > 0.35 -> RWs_torque == 0)
- **Att-seq2seq**:
  always (RWs_angular_mxyomemeEqm < 0.3 -> RWs_torque == 0)
- **Seq2seq**:
  always (WNcAi1iDSDDyD1yD2y171a71aa2345324621) 5 ..... display omitted

Example 2: Whenever Op_Cmd changes to Passive then in response Spd_Act changes to 0 after at most 500 time units.

- **Transformer**:
  always (rise (Op_Cmd == Passive) -> eventually [0 : 500] (rise (Spd_Act == 0)))
- **Att-seq2seq**:
  always (rise (Op_Cmd == Passive) -> not (eventually [0 : 500] (Spd_Act == 0)))
- **Seq2seq**:
  always (rise (PIweD > 1 2 . 3 Q 8 y 5 y D y 6 y 1 y 1 R 1 1 ..... display omitted

Example 3: Whenever V_Mot enters the range [1, 12] then in response starting after at most 100 time units Spd_Act must be in the range [100, 1000].

- **Transformer**:
  always (rise (V_Mot >= 1 and V_Mot <= 12) -> eventually [0 : 100] (Spd_Act >= 100 and Spd_Act <= 1000))
- **Att-seq2seq**:
  always (rise (V_Mot >= 1 and V_Mot <= 12) -> not (eventually [0 : 100] (Spd_Act >= 100 and Spd_Act <= 1000)))
- **Seq2seq**:
  always (rise (p_qHX > 4 Q 3 D a Q a D a m y m a O l Q) ..... display omitted

The extrapolation test shows the poor translation of Seq2seq that is consistent with its low accuracy measured in Table 5. The translation quality of Transformer and Att-seq2seq is much higher. It is however sensitive to how similar the patterns used in the informal requirement are to the ones used in the training data.

In Example 1, Transformer makes the correct translation, while Att-seq2seq fails to copy the identifier and the number, and "greater than" is mistranslated into "<". In Example 2, Transformer tends to add a **rise** operator before the subformula wrapped inside an **F** operator, although in some occasions this is equivalent to the actual intention of the requirement because "changes to" often indicates the significance of a rising edge. On the other side, Att-seq2seq adds a negation operator ¬ in front of the subformula starting with an **F** operator, so the meaning becomes reversed. In Example 3, Transformer translates the requirement correctly while Att-seq2seq makes the same mistake. For the three examples, the Seq2seq model all fails to translate them even in form. It tends to generate lengthy symbols without explicit meaning.

The drop in accuracy is mainly due to the lack of training data, which is low compared to non-synthetic millions of training samples in natural language translation. Data augmentation in NLP is a future promising direction to address this problem and to enrich the diversity of training data.

7 CONCLUSION

We studied the problem of translating CPS natural language requirements to STL, commonly used to formally specify CPS properties by the academic community and practitioners. To address the lack of publicly available natural language requirements, we developed a procedure for automatically generating English sentences from STL formulas. We employed a transformer-based NLP architecture to efficiently train an accurate translator from English to STL. Experiments demonstrated promising results.

While this work focuses on STL specifications and CPS applications, the underlying principles can be applied to other domains and specification formalisms and have a significant positive impact on the field of requirement engineering. Unlike natural languages, formal specifications have a very constrained structure. We believe that this observation can be further explored in the future to develop an even more robust translation mechanism and thus further strengthen requirements engineering methodologies.

REFERENCES

[1] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[2] Oded Maler and Dejan Ničković. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[3] Rani Nelken and Nissim Francez. Automatic translation of natural language system specifications. In *Proc. of CAV'96: the 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 360–371. Springer, 1996.

[4] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ICSE' 99: the 1999 International Conference on Software Engineering*, pages 411–420. ACM, 1999.

[5] Aarne Ranta. Translating between language and logic: What is easy and what is difficult. In *Proc. of CADE-23: the 23rd International Conference on Automated Deduction*, volume 6803 of *LNCS*, pages 5–25. Springer, 2011.

[6] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Adv. Robotics*, 22(12):1343–1359, 2008.

[7] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *Proc. of DATE 2015: the 2015 Design, Automation & Test in Europe*, pages 1677–1682. ACM, 2015.

[8] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Trans. Software Eng.*, 41(7):620–638, 2015.

[9] Andrea Brunello, Angelo Montanari, and Mark Reynolds. Synthesis of LTL formulas from natural language texts: State of the art and research directions. In *Proc. of TIME 2019: the 26th International Symposium on Temporal Representation and Reasoning*, volume 147 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[10] Allen P. Nikora and Galen Balcom. Automated identification of ltl patterns in natural language requirements. In *Proc. of ISSRE 2009: the 20th International Symposium on Software Reliability Engineering*, pages 185–194, 2009.

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

[12] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *Proc. of 2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1677–1682. IEEE, 2015.

[13] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. ARSENAL: automatic requirements specification extraction from natural language. In *Proc. of NFM 2016: the 8th International Symposium on NASA Formal Methods*, volume 9690 of *LNCS*, pages 41–46. Springer, 2016.

[14] Alessandro Fantechi, Stefania Gnesi, Gioia Ristori, Michele Carenini, Massimo Vanocchi, and Paolo Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods Syst. Des.*, 4(3):243–263, 1994.

[15] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul Schermerhorn. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *Proc. of ICRA 2009: the IEEE International Conference on Robotics and Automation*, pages 4163–4168, 2009.

[16] Christopher B. Harris and Ian G. Harris. Generating formal hardware verification properties from natural language documentation. In *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, pages 49–56, 2015.

[17] Tainã Santos, Gustavo Carvalho, and Augusto Sampaio. Formal modelling of environment restrictions from natural-language requirements. In *Proc. of SBMF 2018: the 21st Brazilian Symposium on Formal Methods: Foundations and Applications*, volume 11254 of *LNCS*, pages 252–270. Springer, 2018.

[18] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proc. of ICSE: the 27th International Conference on Software Engineering*, ICSE '05, page 372–381, New York, NY, USA, 2005. ACM.

[19] Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[20] Oded Maler and Dejan Ničković. Monitoring properties of analog and mixed-signal circuits. *STTT*, 15(3):247–268, 2013.

[21] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. Signal-based properties of cyber-physical systems: Taxonomy and logic-based characterization. *J. Syst. Softw.*, 174:110881, 2021.

[22] Bardh Hoxha, Houssam Abbas, and Georgios E. Fainekos. Benchmarks for temporal logic requirements for automotive systems. In Goran Frehse and Matthias Althoff, editors, *Proc. of ARCH@CPSWeek 2014/15: the 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 25–30. EasyChair, 2014.

[23] Sempre: Semantic parsing with execution, Accessed 2021.

[24] Rohit J. Kate and Raymond J. Mooney. Using string-kernels for learning semantic parsers. In Nicoletta Calzolari, Claire Cardie, and Pierre Isabelle, editors, *Proc. of ACL 2006: the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, 2006.

[25] Sippycup, Accessed 2021.

[26] Yuk Wah Wong and Raymond J. Mooney. Learning for semantic parsing with statistical machine translation. In Robert C. Moore, Jeff A. Bilmes, Jennifer Chu-Carroll, and Mark Sanderson, editors, *Proc. of Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. The Association for Computational Linguistics, 2006.

[27] Yoav Artzi. Cornell SPF: Cornell semantic parsing framework, 2016.

[28] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017.

[29] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73—-84, September 2014.

[30] Lappoon R. Tang and Raymond J. Mooney. Automated construction of database interfaces: Intergrating statistical and relational learning for semantic parsing. In *Proc. of 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141. Association for Computational Linguistics, 2000.

[31] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proc. of AAAI 96, IAAI 96:the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1050–1055. AAAI Press / The MIT Press, 1996.

[32] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[33] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *Proc. of ASE 2015: the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 574–584. IEEE Computer Society, 2015.

[34] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Kôiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proc. of LREC 2018: the Eleventh International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA), 2018.

[35] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China, 2015. Association for Computational Linguistics.

[36] Arianna D'Ulizia, Fernando Ferri, and Patrizia Grifoni. A survey of grammatical inference methods for natural language learning. *Artif. Intell. Rev.*, 36(1):1–27, 2011.

[37] Christopher B. Harris and Ian G. Harris. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In Luca Fanucci and Jürgen Teich, editors, *Proc. of DATE 2016: Design, Automation & Test in Europe Conference & Exhibition*, pages 966–971. IEEE, 2016.

[38] Bradford Starkie. Inferring attribute grammars with structured data for natural language processing. In Pieter W. Adriaans, Henning Fernau, and Menno van Zaanen, editors, *Grammatical Inference: Algorithms and Applications, 6th International Colloquium: ICGI 2002*, volume 2484 of *LNCS*, pages 237–248. Springer, 2002.

[39] Nikhil Ketkar. *Introduction to PyTorch*, pages 195–208. Apress, Berkeley, CA, 2017.

[40] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proc. of OSDI 2016: the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pages 5998–6008, 2017.

[42] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. Signal-based properties of cyber-physical systems: Taxonomy

and logic-based characterization. *Journal of Systems and Software*, 174:110881, 2021.

[43] Bardh Hoxha, Houssam Abbas, and Georgios E Fainekos. Benchmarks for temporal logic requirements for automotive systems. *ARCH@ CPSWeek*, 34:25–30, 2014.

[44] Xiaoqing Jin, Jyotirmoy V Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain control verification benchmark. In *Proceedings of the 17th international conference on Hybrid systems: computation and control*, pages 253–262, 2014.

[45] Christoph Gladisch, Thomas Heinz, Christian Heinzemann, Jens Oehlerking, Anne von Vietinghoff, and Tim Pfitzer. Experience paper: Search-based testing in automated driving control applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 26–37. IEEE, 2019.

[46] Zhiyu Liu, Bo Wu, Jin Dai, and Hai Lin. Distributed communication-aware motion planning for multi-agent systems from stl and spatel specifications. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4452–4457. IEEE, 2017.

[47] Parv Kapoor, Anand Balakrishnan, and Jyotirmoy V Deshmukh. Model-based reinforcement learning from signal temporal logic specifications. *arXiv preprint arXiv:2011.04950*, 2020.

[48] Derya Aksaray, Austin Jones, Zhaodan Kong, Mac Schwager, and Calin Belta. Q-learning for robust satisfaction of signal temporal logic specifications. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 6565–6570. IEEE, 2016.

[49] Hsuan-Cheng Liao. A survey of reinforcement learning with temporal logic rewards, 2020.

[50] Wenliang Liu and Calin Belta. Model-based safe policy search from signal temporal logic specifications using recurrent neural networks. *arXiv preprint arXiv:2103.15938*, 2021.

[51] Gang Chen, Mei Liu, and Zhaodan Kong. Temporal-logic-based semantic fault diagnosis with time-series data from industrial internet of things. *IEEE Transactions on Industrial Electronics*, 68(5):4393–4403, 2020.

[52] Infenion datasheet. https://www.infineon.com/dgdl/Infineon-BTS5016-2EKA-DS-v01_00-EN.pdf?fileId=5546d4625a888733015aa41a5e161129.

[53] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.

[54] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[55] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[56] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640. Association for Computational Linguistics, 2016.

[57] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. of ACL 2016: the 54th Annual Meeting of the Association for Computational, Volume 1: Long Papers*, 2016.

[58] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[59] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[60] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[61] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[62] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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