Systematic Translation from Natural Language Robot Task Descriptions to STL

Sara Mohammadinejad[1], Sheryl Paul[1], Yuan Xia[1], Vidisha Kudalkar[1], Jesse Thomason[1], and Jyotirmoy V. Deshmukh[1]

University of Southern California

Abstract. Natural language is an intuitive way for humans to communicate formal requirements of cyber-physical systems, such as safety specifications, performance requirements, and task objectives with autonomous cyber-physical systems such as robots. While *natural language* (NL) is ambiguous, real world tasks and their safety requirements need to be communicated unambiguously. Signal Temporal Logic (STL) is a formal logic that can serve as a versatile, expressive, and unambiguous *formal* language to describe robotic tasks. On one hand, existing work in using STL for the robotics domain typically requires end-users to express task specifications in STL, which is a challenge for non-expert users. On the other, translating from NL to STL specifications is currently restricted to specific fragments. In this work, we propose DialogueSTL, an explainable and interactive approach for learning correct and concise STL formulas from (often) ambiguous NL descriptions. We use a combination of semantic parsing, pre-trained transformer-based language models, and user-in-the-loop clarifications aided by a small number of user demonstrations to predict the best STL formula to encode NL task descriptions. An advantage of mapping NL to STL is that there has been considerable recent work on the use of reinforcement learning (RL) to identify control policies for robots. We show we can use Deep Q-Learning techniques to learn optimal policies from the learned STL specifications. We demonstrate that DialogueSTL is efficient, scalable, and robust, and has high accuracy in predicting the correct STL formula with a few number of demonstrations and a few interactions with an oracle user.

Keywords: Natural language · Signal temporal logic · Robotics.

1 Introduction

Future human societies are likely to interact with general purpose robots using natural language commands. Unfortunately, natural language descriptions can be ambiguous and can have multiple meanings or under-specify the task. For example, consider the command "If it is dark, turn on the lamp before picking up the book.": this leaves it up to the robot to interpret it as "if it is dark, turn on the lamp and pick up the book" (i.e. do nothing if it is not dark), or, "if it is dark, turn on the lamp and then pick up the book, else pick up the book" Similarly, with a command like- "pick up the door key card and open the door". A robot would need to infer under-specified information such as, how

soon should the door be opened, how long after the key card is picked up should the robot wait, which door to open, etc. Signal Temporal Logic (STL) [15] has been used as a flexible, expressive, and unambiguous language to describe robotic tasks that involve time-series data and signals. For instance, STL can be used to formulate properties such as "the robot should immediately extinguish fires but can accept delays in opening doors." From a grammar-based perspective, an STL formula can be viewed as atomic formulas combined with logical and temporal operators [17]. In this work, we match different components of a natural language description with atoms and operators to form candidate STL formulas, and use dialogue with users to resolve ambiguities.

Formalizing behavior using temporal logics such as STL requires the user to specify the correct temporal logic specification [4,19]—a difficult and error-prone task for untrained human users. Translating a sentence written in a natural and ambiguous language into a more general and concise formal language is an open challenge [12].

Seminal works in [11] considers the problem of interactive synthesis of agent policies from natural language specifications. The authors consider similar environments as ours but expect specifications to be provided in structured English that is then translated to LTL formulas. The main objective is to explore the space of specifications using user demonstrations and constraint-based methods to identify the precise specifications, which are then used to synthesize robot policies using reactive synthesis methods. In our work, the emphasis is on directly learning the structure of the task objective using modern natural language processing tools and employing a dialogue-based method to refine the specification. Furthermore, we use recently developed RL methods to learn optimal policies from STL instead of LTL synthesis approaches.

Finally, we would like to provide some historical context to this work: this work was initiated before the emergence of large language models like GPT versions higher than 3.5. It is certainly possible to repeat the experiments performed in this paper using state-of-the-art large language models, and preliminary evidence suggests that they perform well in the translation task. Nevertheless, we argue that we are proposing a compositional approach that could help debug wrong formulas, create chain-of-reasoning prompts to refine or repair subformulas, and other specification engineering tasks. We argue that this makes our approach valuable even in a space dominated by LLM-based translation.

Fig. 1 shows the high-level flow of DialogueSTL, whose input consists of natural language description (NL) and a few demonstrations (demos) of a robotic task.

Our framework involves training a Dual Intent and Entity Transformer (DIET) [6] model on synthetic datasets to accurately predict STL atoms from verb phrases. Bidirectional Encoder Representations from Transformers (BERT), a state-of-the-art tool for masked language modeling and next sentence prediction, is used to build an operator predictor for identifying appropriate STL operators for natural language conjunctions and adverbs. We then apply part-of-speech tagging with Flair to extract phrases from NL descriptions and construct Parametric STL based on causal and temporal dependencies as clarified by the user. Finally, we select the correct formulas and demonstrate its use in learning opti-

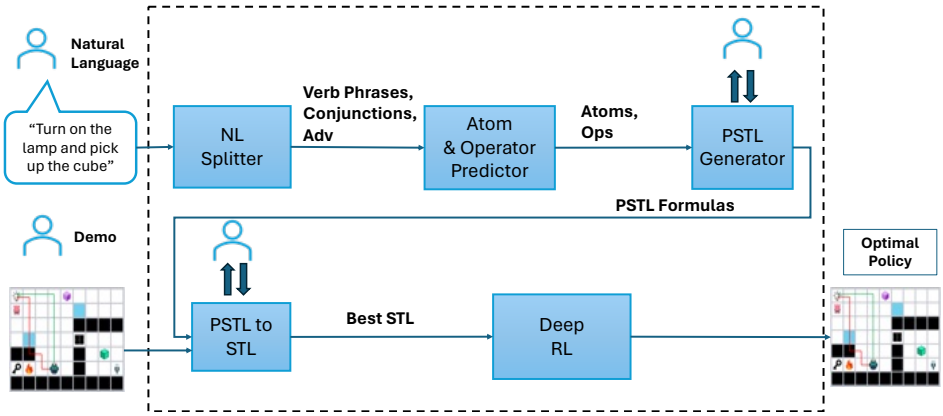

Fig. 1. We infer a STL formula and optimal policy from a given natural language description, a few demonstrations, and questions to the user.

mal policies. We show experimentally that our method is efficient and scalable, and note that in most cases the user has to provide only a few demonstrations—often only one, of a successful behavior for our framework to arrive at the correct STL formula using dialogue interactions.

1.1 Contributions

We summarize the contributions of our approach as follows:

1. We propose DialogueSTL- an interactive and explainable method for learning STL formulas from ambiguous natural language descriptions that additionally provides for user-in-the-loop clarifications where we separately predict atomic signal predicates and logical and temporal operators.
2. We construct Parametric STL (PSTL) formulae based on causal and temporal dependencies - a consideration often overlooked, and use off-the-shelf Deep Q-learning (DQN) to learn optimal control policies for robots from the derived STL specifications (using STL robustness as a reward function).
3. Evaluation on gridworld experiments indicates that we minimize the number of user demonstrations and dialogue interactions to refine and accurately determine STL specifications. Moreover, we also demonstrate superior accuracy and training efficiency compared to DeepSTL, highlighting the transparency provided by DialogueSTL's explanation dictionaries.

2 Background

Definition 1 (Demonstration). *Demonstration is a finite sequence of state-action pairs. Formally,* $\mathbf{d} = \{(\mathbf{s}_0, \mathbf{a}_0), (\mathbf{s}_1, \mathbf{a}_1), ..., (\mathbf{s}_\ell, \mathbf{a}_\ell)\}$ *defines a demonstration with length* ℓ*, where* $\mathbf{s}_i \in S$ *and* $\mathbf{a}_i \in A$*. S is the set of all possible states and A is the set of all possible actions in an environment.*

Definition 2 (Trace). *A trace* $\mathbf{x}$ *is a mapping from time domain* $\mathcal{T}$ *to value domain* $\mathcal{D}$, $\mathbf{x} : \mathcal{T} \to \mathcal{D}$ *where,* $\mathcal{T} \subseteq \mathbb{R}^{\geq 0}$, $\mathcal{D} \subseteq \mathbb{R}^n$, $\mathcal{T} \neq \emptyset$, *and the variable* n *denotes the trace dimension.*

Signal Temporal Logic (STL). Signal Temporal Logic (STL) is a logic to reason about properties of real-valued signals. The basic primitive in STL is called atomic predicate or atom. Atoms are formulated as $f(\mathbf{x}) \sim c$, where $\mathbf{x}$ is a trace, f is a scalar-valued function over the trace $\mathbf{x}$, $\sim \in \{\geq, \leq, =\}$, and $c \in \mathbb{R}$. For instance, $\mathbf{x} \leq 1$ is an atomic predicate, where $f(\mathbf{x}) = \mathbf{x}$, $\sim$ is $\leq$, and $c = 1$. Temporal specifications are created by adding operators such as $\mathbf{G}$ (always), $\mathbf{F}$ (eventually) and $\mathbf{U}$ (until) to atoms. Each temporal operator is indexed by an interval $I := (a, b) \mid (a, b] \mid [a, b) \mid [a, b]$, where $a, b \in \mathcal{T}$ and $a < b$. For example, $\mathbf{G}_{[0,5]}(\mathbf{x} \leq 1)$ means that signal $\mathbf{x}$ is *always* less than or equal to 1 between timesteps 0 to 5. Formally, the STL syntax is defined as follows:

$$\varphi := \mathit{true} \mid f(\mathbf{x}) \sim c \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathbf{U}_I \, \varphi_2, \tag{1}$$

$\mathbf{G}$ and $\mathbf{F}$ operators are special instances of $\mathbf{U}$ operator and can be written as $\mathbf{F}_I\varphi \triangleq \mathit{true} \, \mathbf{U}_I \, \varphi$, and $\mathbf{G}_I\varphi \triangleq \neg\mathbf{F}_I\neg\varphi$.

The Boolean satisfaction of an atomic predicate is *true* if the predicate is satisfied and *false* if it is violated, and the semantics of logical and temporal operators are defined as:

- $\neg\varphi$ is satisfied if φ is not satisfied or φ is violated.
- $\varphi_1 \wedge \varphi_2$ holds if both φ_1 and φ_2 are satisfied.
- $\varphi_1 \vee \varphi_2$ holds if either φ_1 or φ_2 is satisfied.
- $\varphi_1 \rightarrow \varphi_2$ is equivalent to $\neg\varphi_1 \vee \varphi_2$, which means that either φ_1 should not hold or φ_2 should hold.
- $\mathbf{G}_I\varphi$ means φ must hold for all instances of interval I.
- $\mathbf{F}_I\varphi$ means φ must hold at least once in interval I.
- $\varphi_1\mathbf{U}_I\varphi_2$ means φ_1 must hold in I until φ_2 is satisfied.

The *globally* operator $\mathbf{G}_I\varphi$ specifies the property φ must hold true for all time points t' within the interval I starting from t. This means that from the start of the interval t to the end of $t + I$, the condition φ must be continuously held as true. The *eventually* operator $\mathbf{F}_I\varphi$ states that there exists at least one-time point t' within the interval I starting from t where the property φ is true. s is the signal function. The formal definition can be referred as:

$$s, t \models \mathbf{G}_I\varphi \quad \text{iff} \quad \forall t' \in t + I,\, s, t' \models \varphi$$
$$s, t \models \mathbf{F}_I\varphi \quad \text{iff} \quad \exists t' \in t + I,\, s, t' \models \varphi$$

The key aspect of STL is the nest temporal operators within the intervals. This nesting capability enables STL to express more complex temporal relationships. The operator $\mathbf{GF}_I\varphi$ guarantees the satisfaction of the property φ within the time frame, embodying a liveness property. $\mathbf{FG}_I\varphi$ ensures that starting from some point in time t, the property φ will continuously hold true for every subsequent interval I. Formally, they are defined as:

$$s, t \models \mathbf{GF}_I\varphi \quad \text{iff} \quad \forall t \geq 0, \exists t' \in t + I,\, s, t' \models \varphi$$
$$s, t \models \mathbf{FG}_I\varphi \quad \text{iff} \quad \exists t \geq 0, \forall t' \in t + I,\, s, t' \models \varphi$$

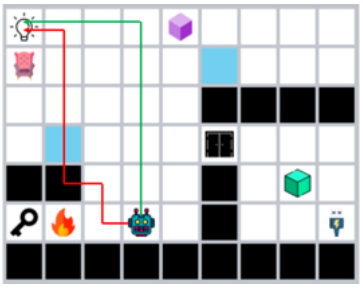

Fig. 2. The robot tries to reach the lamp placed at location $(0, 0)$ in 15 seconds while avoiding wall (black) and water (blue) tiles. Both green (—) and red (—) demonstrations satisfy the formula $\mathbf{F}_{[0,15]}(\mathit{robotAt(0,0)} == 0)$; in the next 15 seconds, the robot should eventually reach to the location $(0, 0)$.

Example 1. Consider the grid world environment illustrated in Fig. 2. While both demonstrations reach the lamp, only the green demonstration satisfies the formulas $\mathbf{G}(\neg(\mathit{robotAtWall} \geq 0))$[1] and $\mathbf{G}(\neg(\mathit{robotAtWater} \geq 0))$. The formula $\mathbf{G}(\neg(\mathit{robotAtWall} \geq 0))$ means that the robot should never climb walls. The formula $\mathbf{G}(\neg(\mathit{robotAtWater} \geq 0))$ means that the robot should not step in water. The red demonstration intersects with both walls and water tiles.

Parametric Signal Temporal Logic (PSTL). A PSTL [2] formula is an extension of an STL formula where constants are replaced by parameters. A STL formula is obtained by pairing a PSTL formula with a valuation function that assigns a value to each parameter variable. For example, consider the PSTL formula $\varphi(x, y, \tau) = \mathbf{F}_{[0,\tau]}(\mathit{robotAt(x,y)} \geq 0)$ with parameters τ, x and y. The STL formula $\mathbf{F}_{[0,15]}(\mathit{robotAt(0,0)} \geq 0)$, which is an instance of φ, is obtained with the valuation $\nu = \{\tau \mapsto 15, x \mapsto 0, y \mapsto 0\}$. From a grammar-based perspective, a PSTL formula can be viewed as atomic formulas combined with unary or binary operators [17].

$$\begin{aligned} &\varphi := \mathit{atom} \mid \mathit{unaryOp}(\varphi) \mid \mathit{binaryOp}(\varphi, \varphi) \\ &\mathit{unaryOp} := \neg \mid \mathbf{F}_I \mid \mathbf{G}_I \\ &\mathit{binaryOp} := \vee \mid \wedge \mid \mathbf{U}_I \mid \Rightarrow \end{aligned} \tag{2}$$

For instance, PSTL formula $\mathbf{F}_{[0,\tau]}(\neg(\mathit{robotAt(a,b)} \geq 0) \vee \mathit{robotAt(c,d)} \geq 0)$ consist of atoms $\mathit{robotAt(a,b)} \geq 0$ and $\mathit{robotAt(c,d)} \geq 0$, unary operators $\neg, \mathbf{F}$ and binary operator $\vee$.

3 DialogueSTL: Learning PSTL Candidates

In this section, we propose an explainable and interactive approach to learn candidate Parametric STL (PSTL) formulas from the natural language description

[1] *robotAtWall* is a trace that its value at time instance t is computed as distance of the robot from the closest wall at time t.

of a task or constraints provided by the user. The overall structure of our approach is shown in Algo. 1. As a running example, consider the command "turn on the lamp and pick up the cube" for the grid world environment shown in Fig. 2. The high-level view of our method for the running example is illustrated in Fig. 3, and the method for converting NL to candidate PSTL formulas is formalized in Algo. 1. The inputs of the algorithm consist of the NL description of the task (`taskNL`), sample data for each atomic predicate (`sampleAtoms`) and operator (`sampleOps`) that we generate manually, and a threshold ϵ on the confidence of an atom predictor.

We first generate a synthetic dataset for atom predictor (AtomPredictor) using a GPT-3 based paraphrase generator (GPT3ParaphGen) and train a model to predict likely atoms from individual verb phrases.[2] For a given NL description, verb phrases, conjunctions and adverbs are extracted and matched with atoms and operators. Candidate PSTL formulas of bounded lengths are then enumerated using predicted atoms and operators.

We now explain each part of Algo. 1:

Algorithm 1: Natural Language to PSTL algorithm

Input: `taskNL`, `sampleAtoms`, `sampleOps`, ϵ=*0.5*
Output: `PSTLFormulas`
// Generate data for atom predictor
`atoms` := GPT3ParaphGen(`sampleAtoms`)
`trainAtoms`, `testAtoms` := TrainTestSplit(`atoms`)
AtomPredictor := `train`(`trainAtoms`) // Train
Accuracy := `test`(`testAtoms`) // Test
// Avg. embedding for each operator
`opEmbeddings` := computeOpsBertEmbeddings(`sampleOps`)
`taggedTokens` := partOfSpeechTagger(`taskNL`)
// Extract verb phrases, conjunctions and adverbs based on the tags
`vPhrases`, `Conjs`, `Advbs` := NLSplitter(`taggedTokens`)
// Find best atoms
for `vPhrase` $\in$ `vPhrases` **do** // Find best atoms
  `atom`, `confidence` := AtomPredictor(`vPhrase`) **if** `confidence` $\leq \epsilon$ **then**
    `vPhrase` := ParaphrasedByUser(`vPhrase`)
    `atom`, `confidence` := AtomPredictor(`vPhrase`)
// Find best operators
`ops` := findBestOps(`opEmbeddings`, `Conjs`, `Advbs`)
// Bounds on the length of PSTL formula
`l`, `u` := $2\cdot|$`vPhrases`$|-1, 2\cdot|$`vPhrases`$|+|$`Conjs`$|+|$`Advbs`$|$
// Enumerative, interactive PSTL synthesis
`PSTLFormulas` := genPSTLEnum(`atoms`, `ops`, `l`, `u`)

[2] We denote a *verb phrase* loosely as a group of words that contains a verb, such as "turn on the lamp," "if fire is on," and "open the door."

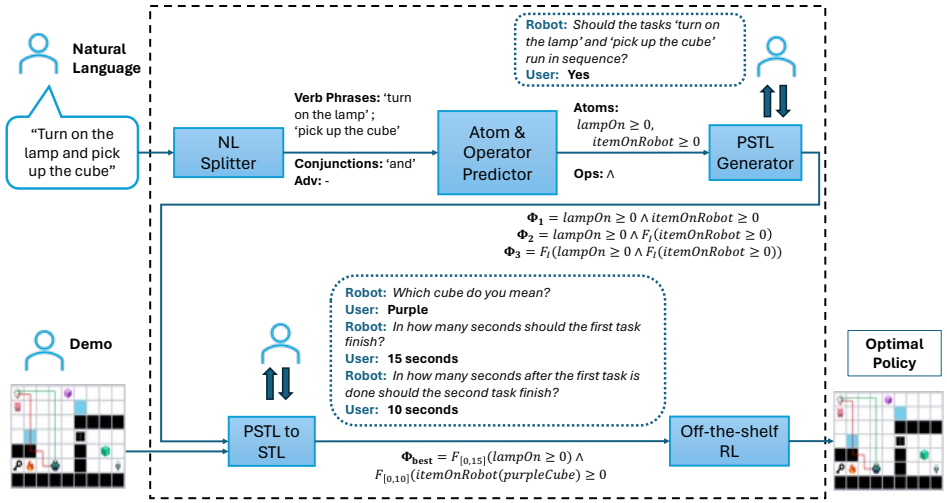

Fig. 3. We infer a STL formula $\varphi_{best} = \mathbf{F}_{[0,15]}(lampOn \geq 0 \wedge \mathbf{F}_{[0,10]}(itemOnRobot(purpleCube) \geq 0))$ and an optimal policy from a given natural language description "Turn on the lamp and pick up the cube", a demonstration (—), and interactions with the user. NL splitter extracts components of the Natural Language (i.e., "Turn on the lamp", "and", "pick up the cube"). Each component is mapped to an atom or operator using the Atom and Operator Predictors {"Turn on the lamp":$lampOn \geq 0$, "and":$\wedge$, "pick up the cube":$itemOnRobot \geq 0$}. Next, candidate PSTL formulas are generated from the predicted atoms and operators. Asking questions from the user can help learn parameters of PSTL formulas and therefore, learning the best STL formula. Finally, Deep RL techniques are employed to learn an optimal policy from the learned STL formula.

Synthetic Data Generation. Translation to STL can prove challenging because of the scarcity of training data. We overcome the brittleness of a language-to-STL predictor based on hand-crafted grammar [12] using GPT-3 [5] as a paraphrase generation tool for data augmentation over a small manual set. This corresponds to lines 1-2, where synthetic data is generated using GPT-3 based paraphrasing for the atom predictor.

Then, we feed manually generated verb phrases corresponding to STL atoms in our grid world as input and check the paraphrasing quality of the GPT-3 outputs. For the grid world demonstrated in Fig. 2, there are a total of 15 atoms, and we generate 108 verb phrase samples for these atoms. For example, our GPT-3-based paraphrase generator gets "Turn off the fire" as input and generates "Extinguish the fire" as one output paraphrase, and both can be paired with the atom *fireOff* as training data.

Atom Predictor. Given a set of verb phrases and their corresponding atomic formulas from the aforementioned data generation, we learn a model using DIET [6], that given a verb phrase, can output the most similar atom to the verb phrase. DIET is a lightweight transformer-based architecture that outperforms

and is about six times faster to train than BERT [9]. We use BERT for operator prediction. DIET employs a Conditional Random Field (CRF) tagging layer on top of the transformer output and a multi-task loss function, which helps in predicting entities in a sequence. We trained DIET for 100 epochs, which took less than 1 minute and resulted in training accuracy of 100% and test accuracy of 92%. We reserve 80% of the data generated by GPT-3 for training and 20% for validation. This part of the algorithm is detailed in lines 3- 4.

Operator Predictor. First, we map each operator with a few words in natural language that correspond to that operator. For example, 'and' and 'and then' both correspond to the $\wedge$ operator. The word embedding of each operator is then computed as the average BERT embedding of the words corresponding to that operator. The most similar operator contains the greatest cosine similarity with each conjunction or adverb. This process is described in lines 5-6.

Natural Language Splitting. To extract verb phrases, we first run a part-of-speech tagger, Flair [1]. It is a state-of-the-art tagger based on contextual string embeddings and neural character language models. We divide the language description based on the position of verbs, resulting in, for example, "turn on the lamp" and "pick up the cube". We extract conjunctions from the words that connect the verb phrases, such as 'and'. any adverbs (for instance, 'Always' can correspond to the globally operator **G**). We try to match each verb phrase with an atom using the trained atom predictor. Each conjunction or adverb is matched with an operator using the cosine similarity between the operators' and words' BERT embeddings. We always add **F** to the list of candidate operators because it is a common operator[3]. In our running example, we extract $atoms = [lampOn \geq 0, itemOnRobot \geq 0]$ and $operators = [\wedge, \mathbf{F}]$. If the confidence of verb phrase to atom correspondence is low (< 0.5), we ask the user to paraphrase the word sequence that has low confidence. This procedure is reflected in lines 7-11.

Explainability. Next, the extracted verb phrases, conjunctions and adverbs are mapped to atoms and operators using the trained Atom and Operator Predictors (12 - 13). This gives us an explanation dictionary that provides clarity on the decisions of DialogueSTL. For our running example: {"Turn on the lamp": $lampOn \geq 0$, "and": $\wedge$, "pick up the cube": $itemOnRobot \geq 0$} is the generated explanation dictionary. This dictionary can help us repair the tool if an incorrect STL formula is predicted. For example: if the atom $lampOn \geq 0$ is incorrectly predicted as $fireOn \geq 0$, we will need to improve the robustness of Atom Predictor. This is one of the advantages of our tool compared to DeepSTL, which is completely black-box.

PSTL Generation. To enumerate possible PSTL formulas, we must specify the upper and lower bounds of their lengths. We consider the lower bound as $l = |vPhrases| + |vPhrases| - 1$ because n verb phrases need $n - 1$ connectors. The upper bound is $u = 2 \cdot |vPhrases| + |Conj| + |Adv|$- we multiply $|vPhrases|$ by 2 because each verb phrase can require a **F** operator; each conjunction or adverb might also be converted to an operator. Next, we use systematic PSTL

[3] Users tend not to specify an explicit word that corresponds to the eventually operator **F**, even if they do expect the robot to perform the task eventually.

enumeration [17] to generate candidate PSTL formulas within the range l and u using the extracted atoms and operators in increasing order of their length. We remove enumerated PSTL formulas that do not contain all the atoms or contain more than one instance from each atom. 14

For instance, in section 2, the PSTL formula $\mathbf{F}_{[0,\tau]}(\neg(\mathit{robotAt(a,b)} \geq 0) \vee \mathit{robotAt(c,d)} \geq 0)$ can be derived through an enumeration process. This process utilizes the atoms $\mathit{robotAt(a,b)} \geq 0$ and $\mathit{robotAt(c,d)} \geq 0)$ where a, b, c, d serve as parameters. The formula incorporates both unary and binary operators to construct the logical expression.

Causal and Temporal Dependency. We shrink the space of candidate PSTL formulas using the idea of causal or temporal dependency between atoms. Atom_2 is causally dependent on Atom_1 if Atom_1 should happen before Atom_2, which eliminates formulas such as $\mathbf{F}(\mathit{Atom}_1) \wedge \mathbf{F}(\mathit{Atom}_2)$ and $\mathit{Atom}_2 \wedge \mathbf{F}(\mathit{Atom}_1)$. In our example, we can ask the user whether the tasks "turn on the lamp" and "pick up the cube" should run in sequence or not. Alternatively, we could ask user if "turn on the lamp and then pick up the cube" is acceptable or not. If the answer is "Yes", this means that the atom $\mathit{itemOnRobot} \geq 0$ is causally dependent on $\mathit{lampOn} \geq 0$, and hence, formulas $\mathbf{F}(\mathit{lampOn} \geq 0) \wedge \mathbf{F}(\mathit{itemOnRobot} \geq 0)$ and $\mathit{itemOnRobot} \geq 0 \wedge \mathbf{F}(\mathit{lampOn} \geq 0)$ would be omitted. In this specific example, the reasoning for causal dependency is that if room is dark, it would be difficult for the robot to identify the cube and pick it up. From the nine generated PSTL formulas, six are removed resulting in three remaining candidate PSTL formulas.

$$\varphi_1 = \mathit{lampOn} \geq 0 \wedge \mathit{itemOnRobot} \geq 0$$
$$\varphi_2 = \mathit{lampOn} \geq 0 \wedge \mathbf{F}_I(\mathit{itemOnRobot} \geq 0)$$
$$\varphi_3 = \mathbf{F}_I(\mathit{lampOn} \geq 0 \wedge \mathbf{F}_I(\mathit{itemOnRobot} \geq 0))$$

4 DialogueSTL: Selecting Correct STL

We find the parameters of the PSTL candidates by searching in the initial language description and by interacting with the user (Algo. 2). We use 4 question types: order of tasks, atom parameters, operator parameters, and for paraphrasing a verb phrase.

DialogueSTL takes positive and negative demonstrations as input. Asking the user to generate many demonstrations makes for a tedious interface. We use only a few demonstrations from the user and use those to automatically generate more negative demonstrations.

To generate negative examples, we use the principle of "no excessive effort" [11], which means that any prefix of the demonstrated positive example is assumed insufficient and considered a negative example. The intuition is that if a prefix of $\mathbf{d}$ would already be a good example, then the user would not have given the full demo $\mathbf{d}$. In our example if the full demonstration is the robot turning on the lamp and then picking up the cube, a prefix where the robot only turns on the lamp without picking up the cube would be considered a negative example. This approach ensures that partial behaviors that do not meet

the full requirements are marked as negative, helping refine the STL formula to accurately capture the desired behavior.

To convert each PSTL formula to its corresponding STL formula, all parameter values must be discovered. For instance, operator **F** needs a time interval and atom $itemOnRobot \geq 0$ requires the name of the item that should be picked up. We search for parameter values in NL description, and if any of the parameter values cannot be found, we find it by interaction with user. We replace the parameter valuations in PSTL formulas which result in STL candidates. Finally, we choose the STL formula that satisfies positive demos and does not satisfy the negative demos. In our running example, among the three candidate STL formulas, φ_3 is chosen as the best STL formula.

If no STL formula can be found, it may be that the language description was not correctly split into its components, or that one or more of the predicted atoms and operators were not correct. In the future, we can improve the splitting algorithm by learning from incorrect predictions, and use beam search in enumeration to move to the next highest ranking atom or operator when the first set fails.

For checking whether a demo satisfies a STL formula we have to compute the satisfaction degree or robustness of its atoms and then use Breach [10] to compute the satisfaction of the entire formula. For instance, the satisfaction degree of the atom $lampOn \geq 0$ is computed based the distance of the robot to lamp and the action of turning the lamp on after the robot reaches the lamp.

5 Learning optimal policies

Previous works have used the robustness of an STL formula, or the signed distance of a given trajectory from satisfying or violating a given formula, as rewards to guide the RL algorithm [19,4]. Here, we only provide an example of learning optimal policy from a given STL formula using those existing techniques. We use Deep Q-Learning [16] because of scalability to environments with a large number of states. Even in our grid simple world environment (Fig. 2), there are more than 8 billion states.[4]

The algorithm takes the STL specification of the task φ_{task}, the goal state and the hyper-parameters (i.e., M, C, γ, and etc.) as input, and generates the optimal policy that respects φ_{task} as output. The main RL loop runs for $|episodes|$. In each episode, first, the state is set to initial state and the partial trajectory is set to $\emptyset$. While the robot has not reached the final state or maximum number of states is not reached, the robot explores the grid environment and the reward is computed as robustness of the partial trajectory with respect to φ_{task}. The robot experiences are recorded in replay memory to be used later for training the Q network. Whenever the replay buffer size exceeds M, we start training the Q network using the bellman equation. We update the weights of target action-value function $\hat{Q}$ with the weights of Q in every C episodes. For our running

[4] Each state is a tuple of 16 elements consist of robot and each of the items' (door key, green and purple cube) positions, state of the lamp and fire (on or off), and state of the door (open or close).

Algorithm 2: Learning the best STL formula

Input: `taskNL`, `atoms`, `ops`, `PSTLFormulas`, `Demos` $\mathbf{d}$
Output: φ_{best}
// Generate more positive, negative demos
$\mathbf{d}^+, \mathbf{d}^- :=$ generateMoreDemos($\mathbf{d}$)
// Finding atoms' parameters
for `atom` $\in$ `atoms` **do**
  `atomParams` := findAtomParams(`taskNL`, `atom`)
  if `atomParams` ***not*** `found` **then**
    `atomParams` := getParamsByInteractionWithUser()
// Finding operators' parameters
for `op` $\in$ `ops` **do**
  `opParams` := findOpParams(`taskNL`, `op`)
  if `opParams` ***not*** `found` **then**
    `opParams` := getParamsByInteractionWithUser()
for $\varphi(p)$ $\in$ `PSTLFormulas` **do**
  // Set parameters of the PSTL formula
  $\varphi(v(p))$:= setParams($\varphi(p)$, `atomParams`, `opParams`)
  // Check if the STL formula $\varphi(v(p))$ satisfies positive demos and does not satisfy negative demos
  if $\varphi(v(p)) \models \mathbf{d}^+$ *and* $\varphi(v(p)) \not\models \mathbf{d}^-$ **then**
    $\varphi_{best} := \varphi(v(p))$
    return φ_{best}
return $\emptyset$

example, with $\varphi_{task} = \mathbf{F}_{[0,15]}(lampOn \geq 0 \wedge \mathbf{F}_{[0,10]}(itemOnRobot(purpleCube) \geq 0))$, the reward converges in less than 15000 episodes, and the learned policy is illustrated in Fig. 4.

6 Experiments

To evaluate DialogueSTL, we qualitatively and quantitatively examine its performance in our gridworld (Fig. 2) on natural language specifications whose underlying STL formulas exhibit various qualities. The software to reproduce the results is available at: https://github.com/saramohammadinejad/DialogueSTL.

Table 1 shows a set of manually curated natural language sentences describing a variety of tasks and user constraints paired with the most frequent STL formulas ultimately predicted by DialogueSTL. We use the GPT-3 based paraphrase generator to generate 142 paraphrases each for the sample sentences in Table 1. We also design an Oracle user to interact with DialogueSTL to answer posed interaction questions. The Oracle user is a simple, rule-based program that provides the correct answer to any given question about the true, underlying STL formula. Table 1 shows the source language input, before paraphrasing, the

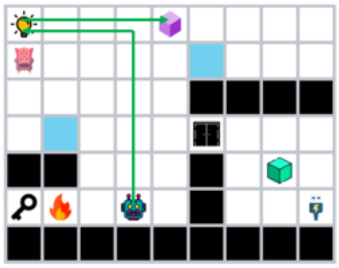

Fig. 4. A demonstration of the learned policy (—) from the $\varphi_{task} = \mathbf{F}_{[0,15]}(lampOn \wedge \mathbf{F}_{[0,10]}(itemOnRobot(purpleCube)))$.

number of provided demonstrations, and the DialogueSTL average number of enumerated formulas, user interactions, success rate, and run-time.[5]

6.1 Results Across Description Types

User Constraints. A human (oracle) provides a general constraint for a task such as "Always don't run into walls", and one positive demonstration that satisfies the constraint and negative one that does not satisfy or violates it. The human user is asked "For how many seconds do you want the constraint to be satisfied?", and answers "1000 seconds", and so the best STL formula is predicted as $\varphi = \mathbf{G}_{[0,1000]}(\neg(robotAtWall \geq 0))$: "Always, in the next 1000 seconds, the robot should not run into walls".

Single Tasks. A human user provides a single task, such as "Pick up the purple cube", and a positive demonstration of the task. Negative examples are generated from this positive example based on the aforementioned principle of 'no excessive effort'. The user is asked about timing requirements, such as: "In how many seconds should the robot complete the task?", and in this case answers "12 seconds". The formula $\varphi = \mathbf{F}_{[0,12]}(itemOnRobot(PurpleCube) \geq 0)$ is predicted, which means "Within the next 12 seconds, the purple cube should be picked up by the robot".

Sequence of Tasks. A sequential task, such as "Go to location (7, 4) and pick up the green cube", requires the robot to do one thing before another—a temporal dependency. The STL formulas that do not give such guarantee can be eliminated from the candidate formulas, and in this case DialogueSTL predicts $\mathbf{F}_{[0,10]}(robotAt(7,4) \geq 0 \wedge \mathbf{F}_{[0,4]}(itemOnRobot(greenCube) \geq 0))$: "In the next 10 seconds, the robot should reach to location (7, 4) and after robot reaches (7, 4), it should pick up the green cube in the next 4 seconds".

Another example of a sequential task is "Turn on the lamp before picking up the purple cube". The word "before" implies the temporal dependency, and the formula predicted is $\mathbf{F}_{[0,12]}((lampOn \geq 0)\mathbf{U}_{[0,8]}(itemOnRobot(purpleCube) \geq 0))$: "Turn on the lamp" happens in the past of "pick up the purple cube".

[5] We run the experiments on an Intel Core-i7 Macbook Pro with 2.7 GHz processors and 16 GB RAM.

Type	—User Input— Pre-paraphrase Language	Natural #Ds	#EFs	#UIs	SR	RT	—DialogueSTL— Most Frequent STL Prediction	Correctness
C	Always don't run into walls.	2	7.81	1.0	90%	5.36	$\neg(\mathbf{F}_{[0,1000]}(robotAtWall))$	✓
	Always do not walk into water.	2	12.72	1.0	90%	3.94	$\neg(\mathbf{F}_{[0,1500]}(robotAtWater))$	✓
S	Pick up the purple cube.	1	2.0	1.09	100%	3.71	$\mathbf{F}_{[0,12]}(itemOnRobot(purpleCube))$	✓
	Turn off the fire.	1	2.0	1.0	90%	3.53	$\mathbf{F}_{[0,5]}(fireOff)$	✓
Q	Open the door and then charge yourself.	1	3.0	3.18	100%	4.12	$\mathbf{F}_{[0,8]}(doorOpen \wedge \mathbf{F}_{[0,6]}(chargerPlugged))$	✓
	Go to location (7, 4) and pick up the green cube.	1	4.16	2.5	58%	4.13	$\mathbf{F}_{[0,10]}(robotAt(7,4) \wedge \mathbf{F}_{[0,4]}(itemOnRobot(greenCube))$	✓
	Turn on the lamp before picking up the purple cube.	1	11.8	2.2	100%	9.50	$\mathbf{F}_{[0,12]}(lampOn\ \mathbf{U}_{[0,8]}\ itemOnRobot(purpleCube))$	✓
	Open the gate before picking up the green cube.	1	10.44	2.11	88%	8.56	$\mathbf{F}_{[0,8]}(doorOpened\ \mathbf{U}_{[0,5]}\ itemOnRobot(greenCube))$	✓
M	Turn on the lamp or turn on the fire.	2	7.0	1.0	100%	4.65	$\mathbf{F}_{[0,12]}(lampOn \vee fireOn)$	✓
	Sit on the chair or pick up the purple cube.	2	7.14	1.14	100%	6.65	$\mathbf{F}_{[0,15]}(robotSittingOnChair \vee itemOnRobot(purpleCube))$	✓
D	If gate is open, close it.	1	32.44	1.0	0%	6.68	$\mathbf{F}_{[0,10]}(doorOpen \implies doorClosed)$	χ
	If fire is on, turn off the fire, else pick up the key.	1	1253.33	1.16	0%	67.68	$\mathbf{F}_{[0,10]}((fireOff \implies fireOn) \implies itemOnRobot)$	χ

Table 1. DialogueSTL performance on sample natural language inputs across 142 GPT-3 paraphrases of the inputs for fixed user demonstrations per row (#Ds). We report the average number of enumerated formulas (#EFs), average user interactions (#UIs) to select a final formula, success rate (SR) of finding the exact match correct formula, and average runtime in seconds(RT). The task types include (C)onstraint, (s)ingle, se(Q)uence, (M)ultiple-choice and con(D)itional. Note that " ≥ 0" is removed from all atoms for brevity.

Multiple-choice Tasks. "Turn on the lamp or turn on the fire" means that the robot is required to complete at least one of the two tasks. In such cases, the user can provide two positive demonstrations showcasing the alternative goals. The STL formula predicted for this example is $\mathbf{F}_{[0,12]}(lampOn \geq 0 \vee fireOn \geq 0)$: "In the next 12 seconds, the lamp should be on or the fire should be on".

Conditional Tasks. "If gate is open, close it" is an example of a conditional task; the robot should accomplish a task only if a condition is satisfied. For conditional tasks, our tool fails to predict the correct STL formulas.

6.2 Comparison with DeepSTL

The main differences between DialogueSTL and DeepSTL are:

– DialogueSTL splits the sentence and learns from its components but DeepSTL learns from the entire sentence.

Tool	Avg SR	Avg ACC	Avg test time (seconds)
DialogueSTL	72%	78%	5.9
DeepSTL	**20%**	**54%**	**0.17**

Table 2. Success rate (SR) and accuracy (ACC) comparison of DialogueSTL and DeepSTL on natural language inputs across 142 GPT-3 generated paraphrases.

- DialogueSTL needs a few demonstrations for finding the best STL formula, but DeepSTL only needs NL description of the task.
- DeepSTL is a fully black-box model but DialogueSTL generates explanation dictionaries.

Here, we show that DialogueSTL outperforms DeepSTL in terms of training runtime and accuracy with the cost of increased testing runtime. DialogueSTL's training data is a total of 108 verb phrases for 15 atoms and 18 adverbs and conjunctions for the 7 operators. Since DeepSTL needs complete sentences for training, we use an enumerative approach by systematically applying production rules to enlist valid sentences from the DialogueSTL's training data which results in 250000 sentences with their corresponding STL formulas We randomly sample 120000 instances (the dataset size used in DeepSTL paper), and split it to 80% for training, 10% for validation, and 10% for testing. We train the DeepSTL tool for 60 epochs on our generated training data with exact same transformer structure and hyper-parameters used in the DeepSTL paper. We also tried different sets of hyper-parameters but the default hyper-parameters used in the DeepSTL paper resulted in the best accuracy. The training process takes 23.64 hours resulting in the train, validation and test accuracy of 97.7%, 97.6% and 66.0%, respectively. It is possible to decrease the training time to several hours by using GPUs but still the training time is not comparable with DialogueSTL's which is less than 1 minute. The reason for discrepancy between train/validation and test accuracy might be the use of teacher forcing technique during train/validation.

Next, we test the trained model on GPT-3 generated paraphrases to make a comparison with our tool. We use two metrics for comparing the performances: (1) success rate (SR) which is the percentage of correctly predicted STL formulas (2) accuracy (ACC) which measures how similar the predicted STL formula is to ground truth STL formula. DialogueSTL outperforms DeepSTL in both SR and ACC with the cost of increased test time. The detailed results are presented in Table 2. The reason for large test time is that DialogueSTL only learns Atom Predictor and Operator Predictor during training phase and the formula itself should be learned during testing. Splitting the sentence to its components, learning the formula parameters, enumerating candidate STL formulas and choosing the one that satisfies user's demonstrations are the steps that happen during testing.

6.3 Limitations & Future work

DialogueSTL currently lacks a large scale dataset for NL-TL transformation such as presented in [8] limiting generalizability across domains. While it provides for user-in-the-loop clarifications and feedback and it does not yet incorporate more sophisticated error corrections and mechanisms such as autoregressive re-prompting [7] which can hinder accuracy. Furthermore, we note that DialogueSTL has not yet extended to practical implementations and robots in diverse environments and future work will aim to expand its utility to broader and more varied contexts.

For the conditional task "If fire is on, turn off the fire, else pick up the key", DialogueSTL fails to predict a satisfying formula. The ground truth STL formula for this sentence is $\mathbf{G}((\mathit{fireOn} \geq 0 \implies \mathbf{F}(\mathit{fireOff} \geq 0)) \wedge (\mathit{fireOff} \geq 0 \implies \mathbf{F}(\mathit{itemOnRobot(key)} \geq 0))$. DialogueSTL fails to discover this formula because there are no key words in the sentence to imply the $\mathbf{G}$ and $\wedge$ operators. In the future, gathering such failure cases to create augmented data may fill such gaps. Further, user questions could be designed to recover from such missing operator corner cases. Relatedly, for the task "if fire is off, turn it on", DialogueSTL incorrectly predicts $\mathbf{F}_{[0,8]}(\mathit{fireOff} \geq 0 \implies \mathit{lampOn} \geq 0)$. The atom $\mathit{lampOn} \geq 0$ is selected instead of $\mathit{fireOn} \geq 0$ because "turn it on" does not explicitly say "turn the fire on". Co-reference resolution steps could mitigate this issue.

7 Related work & Conclusions

Related work. Prior work has used STL for reinforcement learning applications. Quantitative semantics of STL can be used as reward functions for evaluating robotic behaviors [4]. STL formulas can be used to rank the quality of demonstrations in robotic domain and also computing the reward for RL problems [19]. However, those works put the burden of specifying the correct STL formulas on users, and can require $3x$ more demonstrations than DialogueSTL despite using a similar environment [19].

There has been a tremendous effort in learning temporal logic languages from natural human languages [13,18,20]. These works variously assume a particular format for natural language, are limited to a specific fragment of formal logic, or have scalability and robustness issues. The authors of [3] provide a comprehensive unified framework that includes qualitative, real-time, and probabilistic property specification patterns. Our approach, DialogueSTL, is interactive and explainable, making it more accessible to non-experts. LtlTalk proposed by [11] focuses on linear temporal logic (LTL) and uses optimization modulo theories for generating LTL specifications and leverages a domain-specific language to expand its expressiveness. While they focus on a single example trace, ours requires more demonstrations and clarifications from the user. However, we also demonstrate practical applications for the synthesized STL using reinforcement learning to learn optimal policies. DeepSTL [12] is another method, that relies on grammar-based generation of synthetic data and transformer models for translating English to STL, while DialogueSTL incorporates user-in-the-loop

clarifications and demonstrations to improve the accuracy and practicality of the generated STL formulas. Our work addresses these shortcomings by operating over the space of all possible STL formulas, leveraging interaction with the user to repair ambiguities, and scaling to a larger state space.

There is recent work [8] focusing on developing an accurate and generalizable framework for transforming NL instructions into temporal logic by creating a large dataset of NL-TL pairs and fine-tuning T5 models achieving high accuracy with minimal training data. There is also work [7] centered on enhancing human-robot interactions by translating NL task descriptions into an intermediate task representation using few-shot learning and Task-and-Motion Planning (TAMP) algorithms, significantly improving task completion through error correction techniques. Lang2LTL [14] grounds navigational commands to LRL using LLMs without prior language data, demonstrating state-of-the-art generalization in diverse environments and practical implmentations on a physical robot. DeepSTL [12], translates informal English requirements to STL by training a sequence-to-sequence transformer on *synthetic* data generated from a hand-defined STL-to-English grammar. DeepSTL is restricted to a specific fragment of STL covered by the hand-defined grammar. For example, it allows the conjunction and disjunction of only two atomic propositions, and some nested formulas are not supported. In Section 6, we show that the proposed DialogueSTL achieves a better accuracy compared to DeepSTL and is significantly faster to train. Furthermore, previous approaches are mostly black-box approaches while DialogueSTL generates explanation dictionaries that gives transparency into the decision of the model, i.e., how different parts of the sentence are mapped to different components of the predicted STL formula.

Conclusions. In this work, we have proposed an interactive and explainable approach, DialogueSTL, to learn STL formulas from natural language descriptions of robotic tasks and demonstrations. We used part-of-speech tagging to extract sentence components and used the GPT-3 language model to generate data automatically given a small sample of manually generated data. Then, transformer models are used for detecting the best atoms and operators for generating candidate PSTL formulas. Finally, demonstrations provided by the user can help learn the best STL formula for a given task. Our tool has a number of advantages compared to previous works such as addressing ambiguity of natural language by interaction with user, considering the space of all STL formulas, and explainability.

References

1. Akbik, A., Bergmann, T., Blythe, D., Rasul, K., Schweter, S., Vollgraf, R.: Flair: An easy-to-use framework for state-of-the-art nlp. In: NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations). pp. 54–59 (2019)
2. Asarin, E., Donzé, A., Maler, O., Nickovic, D.: Parametric identification of temporal properties. In: International Conference on Runtime Verification. pp. 147–160. Springer (2011)

3. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015)
4. Balakrishnan, A., Deshmukh, J.V.: Structured reward shaping using signal temporal logic specifications. International Conference on Intelligent Robots and Systems (IROS) (2019)
5. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020)
6. Bunk, T., Varshneya, D., Vlasov, V., Nichol, A.: Diet: Lightweight language understanding for dialogue systems. arXiv preprint arXiv:2004.09936 (2020)
7. Chen, Y., Arkin, J., Dawson, C., Zhang, Y., Roy, N., Fan, C.: Autotamp: Autoregressive task and motion planning with llms as translators and checkers (2024)
8. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: Transforming natural languages to temporal logics using large language models. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. pp. 15880–15903. Association for Computational Linguistics, Singapore (Dec 2023). https://doi.org/10.18653/v1/2023.emnlp-main.985, https://aclanthology.org/2023.emnlp-main.985
9. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
10. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: International Conference on Computer Aided Verification. pp. 167–170. Springer (2010)
11. Gavran, I., Darulova, E., Majumdar, R.: Interactive synthesis of temporal specifications from examples and natural language. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–26 (2020)
12. He, J., Bartocci, E., Ničković, D., Isakovic, H., Grosu, R.: From english to signal temporal logic. arXiv preprint arXiv:2109.10294 (2021)
13. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Translating structured english to robot controllers. Advanced Robotics **22**(12), 1343–1359 (2008)
14. Liu, J.X., Yang, Z., Idrees, I., Liang, S., Schornstein, B., Tellex, S., Shah, A.: Grounding complex natural language commands for temporal tasks in unseen environments (2023)
15. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)
16. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)
17. Mohammadinejad, S., Deshmukh, J.V., Puranic, A.G., Vazquez-Chanlatte, M., Donzé, A.: Interpretable classification of time-series data using efficient enumerative techniques. In: Proc. of HSCC. pp. 1–10 (2020)
18. Nelken, R., Francez, N.: Automatic translation of natural language system specifications into temporal logic. In: International Conference on Computer Aided Verification. pp. 360–371. Springer (1996)
19. Puranic, A., Deshmukh, J., Nikolaidis, S.: Learning from demonstrations using signal temporal logic. In: Proceedings of the 2020 Conference on Robot Learning (2021)
20. Ranta, A.: Translating between language and logic: what is easy and what is difficult. In: International Conference on Automated Deduction. pp. 5–25. Springer (2011)