Interactive Synthesis of Temporal Specifications from Examples and Natural Language

IVAN GAVRAN, Max Planck Institute for Software Systems, Germany
EVA DARULOVA, Max Planck Institute for Software Systems, Germany
RUPAK MAJUMDAR, Max Planck Institute for Software Systems, Germany

Motivated by applications in robotics, we consider the task of synthesizing linear temporal logic (LTL) specifications based on examples and natural language descriptions. While LTL is a flexible, expressive, and unambiguous language to describe robotic tasks, it is often challenging for non-expert users. In this paper, we present an interactive method for synthesizing LTL specifications from a *single* example trace and a natural language description. The interaction is limited to showing a small number of behavioral examples to the user who decides whether or not they exhibit the original intent. Our approach generates candidate LTL specifications and distinguishing examples using an encoding into optimization modulo theories problems. Additionally, we use a grammar extension mechanism and a semantic parser to generalize synthesized specifications to parametric task descriptions for subsequent use. Our implementation in the tool LtlTalk starts with a domain-specific language that maps to a fragment of LTL and expands it through example-based user interactions, thus enabling natural language-like robot programming, while maintaining the expressive power and precision of a formal language. Our experiments show that the synthesis method is precise, quick, and asks only a few questions to the users, and we demonstrate in a case study how LtlTalk generalizes from the synthesized tasks to other, yet unseen, tasks.

CCS Concepts: • **Software and its engineering** → **Designing software**; • **Human-centered computing** → *Interactive systems and tools.*

ACM Reference Format:
Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive Synthesis of Temporal Specifications from Examples and Natural Language. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 201 (November 2020), 26 pages. https://doi.org/10.1145/3428269

1 INTRODUCTION

As robots move from controlled factory environments to homes, an important challenge is to allow end-users to specify tasks for the robot in clear and unambiguous ways. While there are different formal languages for specifying tasks [Fikes and Nilsson 1971; Ghallab et al. 1998; Kress-Gazit et al. 2009; Levesque et al. 1997], their use is limited to users with programming experience who have mastered the syntax and semantics of the language.

In this paper, we focus on *end-user* programming for robots with linear temporal logic (LTL) as the underlying specification framework. LTL provides a flexible, expressive, and unambiguous mechanism to describe complex tasks, and there are planning and synthesis toolchains from LTL specifications to implementations on robotics platforms [Gavran et al. 2017; Kress-Gazit et al.

Authors' addresses: Ivan Gavran, Max Planck Institute for Software Systems, Kaiserslautern, Germany, gavran@mpi-sws.org; Eva Darulova, Max Planck Institute for Software Systems, Kaiserslautern, Germany, eva@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems, Kaiserslautern, Germany, rupak@mpi-sws.org.

2475-1421/2020/11-ART201
https://doi.org/10.1145/3428269

2009; Lignos et al. 2015; Saha et al. 2016]. Unfortunately, specifying tasks in LTL is challenging for untrained users [Dwyer et al. 1999; Holzmann 2002].

Natural language descriptions and programming-by-examples have been considered as alternative, end-user friendly, ways to specify complex tasks. While LTL specifications can be written in structured natural language [Kress-Gazit et al. 2008], such systems still require a close understanding of LTL syntax and semantics. Dialog-based interfaces are usually limited to a fixed set of the most common scenarios [Kollar et al. 2013; Thomason et al. 2015]; utterances beyond the pre-programmed tasks are rejected.

On the other hand, current programming-by-example techniques for temporal specifications (such as automata or LTL) expect many positive and negative examples to be provided [Cassel et al. 2016; Cobleigh et al. 2003; Neider and Gavran 2018]. Previous work in programming through examples has shown that it is unreasonable to expect end-users to provide more than a few examples [Singh and Gulwani 2015]. Moreover, classical results in learning theory [Angluin 1987; Gold 1967] show that one cannot identify non-trivial classes of formal languages (which includes LTL)—even in the limit—employing only positive examples. Providing negative examples is challenging; while it is simple to show an execution that accomplishes the task at hand, users are confused when asked to show *how not to* accomplish the task. Finally, there is no technique to learn *parameterized* specifications which represent classes of specifications. Thus, end-user programming using LTL as the underlying framework is an open problem.

In this paper, we apply synthesis from examples and natural language utterances to the problem of learning robotic tasks expressed in co-safe LTL formulas. Our approach consists of three components. First, a synthesis procedure that takes a natural language description of a task and an example execution trace from the user and generates a set of candidate LTL specifications. Second, an interactive loop that uses distinguishing examples to identify the correct LTL specification. Third, a generalization step that eventually learns a parameterized LTL specification. The three components ensure the following properties.

- Our approach requires only a single example to be provided by the user, and a few rounds of interaction based on judging examples provided by the system.
- We do not require the user to provide negative examples.
- Our technique generalizes from individual examples to learn a parameterized representation and does not require the user to repeat the synthesis for small changes in tasks.

We bias the search in the synthesis procedure based on the natural language description, and narrow down the search space based on the following two assumptions about user-provided examples. First, we assume a principle of *no excessive trace*: when a user provides an example trace, we assume that no proper prefix of the trace is sufficient to satisfy the underlying specification. Second, we assume a principle of *no excessive effort*: when a user provides an example, we assume that every primitive action in the example is necessary to satisfy the specification, or contrapositively, that the same example with a strict subset of primitive actions does not satisfy the specification. Thus, we can generate a set of negative examples from a single user-provided example.

We learn a set of candidate LTL specifications that are consistent with the one positive example and the generated negative examples, and are informed by the natural language description. We encode the learning problem as a *multi-objective* optimization problem modulo SMT [Bjørner et al. 2015]. The optimization objectives approximate the correspondence of the natural language description to the prospective specification.

Given two candidate formulas, we use another instance of multi-objective optimization modulo SMT to generate a world and a robot behavior that distinguishes between the two candidates (assuming such a world exists). We interactively show the generated behavior to the user and ask

them to judge whether it is consistent with their original command. In this way, we can narrow down candidate solutions to a single one.

While our synthesis procedure learns a specification, one expects that the system maintains a parameterized mapping from natural language descriptions to specifications. For example, it is unreasonable that the user must separately teach the robot to pick up a red item and to pick up a green item. Our system generalizes the learned LTL specification through a form of grammar expansion based on the work in language naturalization [Wang et al. 2017c].

We start with a core DSL to express LTL properties and expand the DSL with new grammar rules whenever the synthesis procedure learns a new mapping. This expansion process, called *naturalization* in the natural language processing literature, allows the system to generalize from previously synthesized specifications and combine them in new ways at a later point. Potential ambiguous parses introduced by the new rules are ranked by a probabilistic model that gets updated through interaction with the user.

We have implemented the synthesis technique in LtlTalk, which implements the synthesis procedure and a planning toolchain on top of a visual interface for a robot navigating a blocks world. LtlTalk's interface allows the user to provide natural language descriptions of tasks as well as example executions in the blocks world. An important characteristic of our domain is that we can provide quick visual feedback to the user showing the effect of a task execution. Thus, we can use the interface to demonstrate distinguishing worlds in the third phase of synthesis.

We have used LtlTalk to learn LTL specifications for a number of common tasks in a simulated world. We empirically show that most tasks can be learned through the use of just one example, a few (less than 4) interactions, and less than 20 seconds. We furthermore show in a case study how naturalization generalizes beyond the synthesized tasks, effectively increasing the scalability of LTL specification synthesis.

Contributions. In summary, the contributions presented in this paper are as follows.

- **One-Shot Synthesis for LTL** An algorithm to create a set of likely candidate specifications from a natural language task specification and one example. The algorithm uses an optimization modulo SMT encoding for LTL specification mining, and augments it by adding cost functions derived from the natural utterance. The synthesis procedure filters candidates interactively. It iteratively devises worlds and plans that distinguish between candidate specifications. The distinguishing process is encoded symbolically and the examples are presented visually to the user.
- **The LtlTalk Task System** We present LtlTalk, that implements the synthesis technique to interactively learn tasks for a robot navigating a blocks world. LtlTalk provides a natural interface to specify tasks, learns the underlying LTL specification, and interfaces with planning and reactive synthesis tools to generate actual plans. The LtlTalk's codebase[1] and web interface[2] are publicly available.

2 OVERVIEW AND MOTIVATING EXAMPLE

Figure 1 shows the high-level view of LtlTalk. LtlTalk maintains an extensible grammar, mapping natural language utterances into a core language (which corresponds to LTL). When a user issues a (natural language) description of a task, if the description can be parsed into LtlTalk's current core language, a plan is generated and executed. If, on the other hand, LtlTalk cannot parse the command using its current grammar, the system synthesizes an LTL specification for the utterance

[1] https://github.com/mpi-sws-rse/ltltalk-interactive-synthesis
[2] https://ltltalk.mpi-sws.org

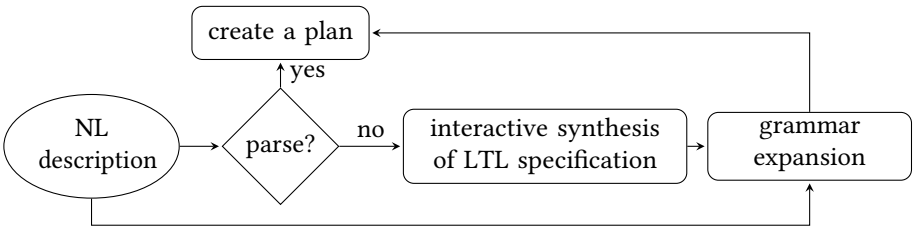


Fig. 1. Schematic view of LtlTalk

through interaction with the user. Having produced the natural language utterance and its LTL equivalent, LtlTalk expands its formal language by inducing new grammar production rules.

Consider the robot simulation world presented in Figure 2. The world includes a robot (the gray cube) that can move about its environment consisting of dry and wet tiles, and pick items of different shapes and colors. LtlTalk internally maintains an extensible grammar, initialized to a version of LTL, that maps known commands to LTL specifications. Consider a situation when a user instructs the robot to *take one red item from (7,4)*, but the robot does not understand the instruction (as this utterance is not yet part of the internal grammar). We show how LtlTalk learns an LTL specification and adds a mapping to its grammar. Initially, LtlTalk asks the user to provide an example for the request by navigating the robot in the simulation world (Figure 2(a)).

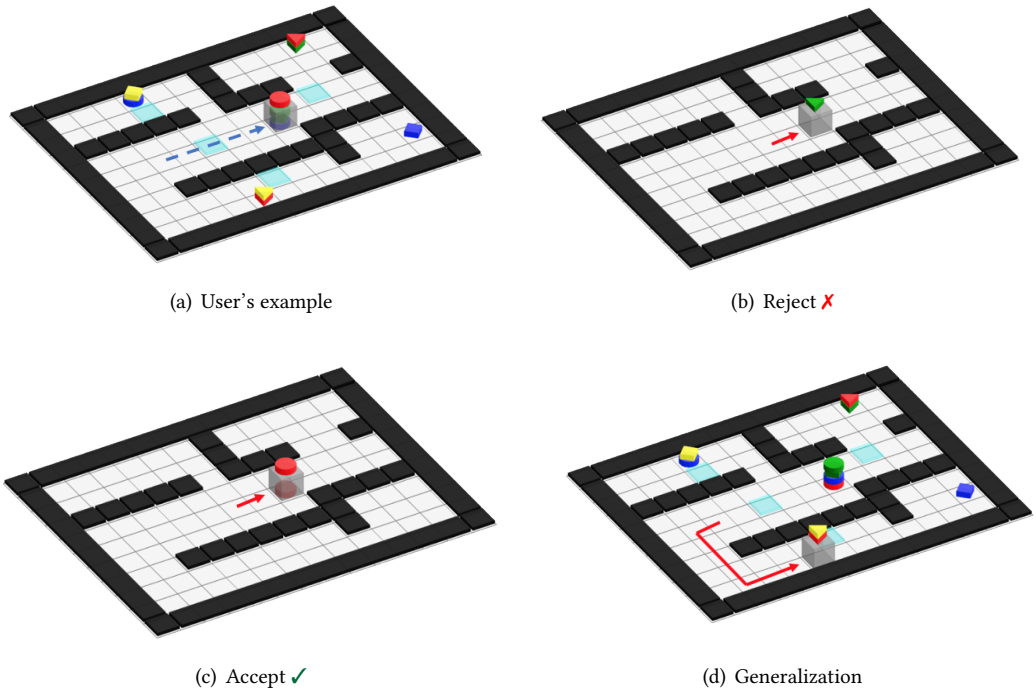
Step 1: Generate Candidates. Using the example provided by the user together with the original natural language instruction, LtlTalk creates a number of candidate specifications:

a) **eventually** pick *one red* item at $(7, 4)$,
b) **eventually** pick *one* item at $(7, 4)$,
c) **eventually** pick *every red* item at $(7, 4)$, and
d) **not** *robot* at *dry* **before** pick *every red* item at $(7, 4)$.

The first three instructions correspond to requesting that eventually one red item, or one item of any color, or every red item from the location $(7, 4)$ is picked; the fourth one corresponds to asking that eventually a red item from the location $(7, 4)$ is picked, but before that the robot must pass over a wet tile. (The full presentation of the core language will be given in Section 5.) Note that there are still other specifications consistent with the example, but the number of generated candidates is limited by a hyperparameter of the algorithm (full details of the algorithm are presented in subsection 4.1).

Step 2: Filter Based on Distinguishing Worlds. In order to determine which one of the candidate specifications is the one that the user had in mind, LtlTalk first shows a world in which there is only one green triangle at the location $(7, 4)$ and the robot picking that item (Figure 2(b)). When the user rejects that behavior, LtlTalk can eliminate b) from the list of candidates. When in the next interaction the user accepts the robot reaching the location $(7, 4)$ without passing over a water tile and picking only one of the two red items (Figure 2(c)), LtlTalk eliminates specifications c) and d), and concludes that the target specification is **eventually** pick *one red* item at $(7, 4)$. (The creation of the distinguishing worlds is described in subsection 4.3.)

Step 3: Generalization and Grammar Extension. Having determined the specification corresponding to the natural language description *take one red item from (7,4)*, LtlTalk uses the (*specification*, *NL description*) pair to form a new, generalized rule. It augments its underlying grammar by the new production rule. (This augmentation is described in subsection 5.2.) With its augmented grammar, LtlTalk is now able to understand expressions such as *take every triangle item from (4,0)*, as illustrated in Figure 2(d).



(a) User's example (b) Reject ✗

(c) Accept ✓ (d) Generalization

Fig. 2. User's interaction with LtlTalk. In 2(a) the user provides an example for the command *take one red item from 7,4* by moving the robot to (7, 4) and picking a red item. Afterwards, based on the example and the (natural language) command, a number of candidate specifications is created. In 2(b) LtlTalk shows the robot picking one green item, which the user rejects. In 2(c) LtlTalk shows the robot picking (7, 4) without ever going through a wet tile and picking only one red item. The user accepts it, thus narrowing down the set of candidates to a single specification. Finally, 2(d) shows that LtlTalk generalized from the inferred specification and now understands commands such as *take every triangle item from 4,0*.

3 FORMAL MODELS FOR TASKS AND THE WORLD

We first formalize the notion of specification language, world model, and user demonstration.

3.1 Linear Temporal Logic

Linear temporal logic (LTL) [Pnueli 1977] is a logic for temporal properties of sequences of events. For a finite set of propositional variables Q, formulas in LTL are defined inductively by a) each $q \in Q$ is an LTL formula, b) if ψ_1 and ψ_2 are LTL formulas, then so are $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\psi_1 \,\mathsf{U}\, \psi_2$, and $\mathsf{X}\,\psi_1$. We treat propositional variables as nullary operators and define $U = \{\neg, \mathsf{X}\}$ and $B = \{\vee, \mathsf{U}\}$ as the set of unary and binary operators, respectively. The set of all operators is called O.

We use a directed acyclic graph as a formula representation (*syntax DAG*). Each node of the syntax DAG is an operator, and depending on its arity, it can have two (binary operators), one (unary operators), or zero (propositional variables) children. Unlike syntax trees, syntax DAGs use only one node per unique subformula. Figure 3 shows a syntax DAG for the formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \vee$

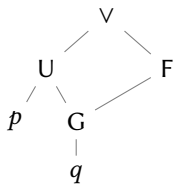


Fig. 3

$\mathsf{F}\,\mathsf{G}\,q$. For a formula ψ, and its set of subformulas $subf(\psi)$, we define the size of ψ to be the cardinality of $subf(\psi)$ (or, equivalently, the number of nodes in its syntax DAG).

We define the semantics of LTL using a valuation function. For a trace $t \in (2^Q)^\omega$, the valuation function $V_t(\tau, \psi)$ for a timestep $\tau \in \mathbb{N}$ and an LTL formula ψ is defined recursively over the subformulas of ψ.

- $V_t(\tau, q) \equiv q \in t[\tau]$, for $q \in Q$
- $V_t(\tau, \neg\psi) = 1 - V_t(\tau, \psi)$
- $V_t(\tau, \psi_1 \vee \psi_2) = \max(V_t(\tau, \psi_1), V_t(\tau, \psi_2))$
- $V_t(\tau, \mathsf{X}\,\psi) = V_t(\tau + 1, \psi)$
- $V_t(\tau, \psi_1 \,\mathsf{U}\, \psi_2) = \max_{i \geq \tau}\{\min[V_t(i, \psi_2), \min_{\tau \leq j < i}(V_t(j, \psi_1))]\}$

We say that a trace t *satisfies* a formula ψ if $V_t(0, \psi) = 1$. As syntactic sugar, one can define constants $\top$ and $\bot$, as well as Boolean operators $\rightarrow$ and $\wedge$ in the usual way. Similarly, two more temporal operators are commonly used: *finally*, F, defined by $\mathsf{F}(\psi) \equiv \top \,\mathsf{U}\, \psi$ and *globally*, G, defined by $\mathsf{G}(\psi) \equiv \neg\,\mathsf{F}(\neg\psi)$. Additionally, we define another temporal operator, B (standing for *before*) by $\varphi\,\mathsf{B}\,\psi \equiv \mathsf{F}(\varphi \wedge \mathsf{X}\,\mathsf{F}\,\psi)$. The formulas for which there exists a *good prefix*, i.e., a finite trace whose every extension satisfies the formula, are called *co-safe formulas*.

While LTL formulas are traditionally interpreted over infinite traces, in practical applications (including task-oriented robotic commands), it is often necessary to interpret LTL formulas over finite traces. To facilitate that need, the semantics of LTL can be modified to support evaluation over finite traces, as was done by De Giacomo and Vardi [2013]. We use the finite traces semantics to evaluate candidate formulas on the user-provided finite example traces.

3.2 The World Model

The propositional variables in our LTL formulas will come from a *world model* describing the world and the robot's actions in it. We take the planning approach and partition the propositional variables into *fluents* $\mathcal{F}$ and *actions* $\mathcal{A}$ [De Giacomo and Vardi 2015]. The set of fluents consists of the propositional or integer variables describing facts about the state of the world; a valuation to the fluents defines a state. An action from $\mathcal{A}$ names a transition between states when the corresponding action is performed, and is also encoded by a propositional variable.

A world w is fully described by a tuple $w = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{A}}, \varphi_{\mathcal{F}}, \varphi_{init})$, where $\mathcal{F}$ is the finite set of *fluents*, $\mathcal{A}$ is the finite set of *actions*, the fluent constraint $\varphi_{\mathcal{F}}$ is an LTL formula on $\mathcal{F}$ describing invariants on the world, the action constraint $\varphi_{\mathcal{A}}$ is an LTL formula describing the effect of an action on the fluents, and φ_{init} is an LTL formula over $\mathcal{F}$ describing the initial state of the world.

For our block world with fixed world width w, world height l, set of colors C, and the set of shapes S, the set of fluents $\mathcal{F}$ contains propositional variables describing the state of the world (locations of obstacles, items, the robot, and what the robot is carrying). For ease of readability, we describe our examples using Prolog-like predicates with relations and (bounded) integer variables. Internally, these predicates are compiled into Boolean propositional variables. We omit a full list of fluents and their propositional encoding, but give examples.

We model the blocks world as a set of tiles (i, j), $i \in \{1, \ldots, w\}$, $j \in \{1, \ldots, l\}$. Each tile (i, j) is exactly one of $wall(i, j)$ (obstacle), $dry(i, j)$ (dry tile), or $wet(i, j)$ (wet tile). The predicate $at(x, y)$ denotes the robot is currently at location (x, y) and $items(i, j)(c, s) = k$ indicates there are k items of color c and shape s at location (i, j). The predicate $carry(c, s, k)$ indicates the robot is carrying k items of color c and shape s.

The fluent constraints specify invariants that must hold on all reachable states. For example, by asserting that the robot can only be in a location within the block world which is not a wall, and

that every location is either a wall, a wet tile, or a dry tile:

$$\bigwedge_{i,j} \mathrm{G}\Big(at(i,j) \rightarrow \neg wall(i,j) \wedge (exactlyOne(dry(i,j), wall(i,j), wet(i,j))\Big)$$

(where *exactlyOne* is the propositional formula that ensures exactly one of its inputs is true) or by specifying that the items can be only at locations that are not a wall:

$$\bigwedge_{i,j,c,s,k} \mathrm{G}\Big(items(i,j)(c,s) = k \rightarrow \big(k \geq 0 \wedge (k > 0 \rightarrow \neg wall(i,j))\big)\Big)$$

The initial world φ_{init} defines the locations of wall and wet tiles, the position of the robot, and the location of all the items.

The actions available to the robot are motion in one of the 4 directions and picking a set of items based on its properties. Moving is expressed by the action $mov(x, y)$, with $x, y \in \{-1, 0, 1\}$, and denotes that the robot moved in relative direction. Picking objects is expressed by the action $pick(c, s) = k$ which indicates that k items of color c and shape s are picked. The action constraints state the effect of an action. Typically, the effect of an action is specified as $\mathrm{G}(\varphi_1 \rightarrow \mathrm{X}(a \rightarrow \varphi_2))$, where $a \in \mathcal{A}$ and φ_1 and φ_2 are propositional formulas on the fluent variables.

For example, the constraint

$$\bigwedge_{i,j} \bigwedge_{(x,y) \in \substack{\{(1,0),(-1,0),\\(0,1),(0,-1)\}}} \mathrm{G}(at(i,j) \rightarrow \mathrm{X}(mov(x,y) \rightarrow (at(i+x, j+y) \wedge \neg wall(i+x, j+y))))$$

states that a move changes the position of the robot to the corresponding cell if the target cell is not a wall.

Other action constraints specify, for example, that the robot can execute exactly one action in each step:

$$\mathrm{G}\left[\bigvee_{a \in \mathcal{A}} a \wedge \bigwedge_{\substack{a,a' \in \mathcal{A} \\ a \neq a'}} (\neg a \vee \neg a')\right]$$

Given a world $(\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{A}}, \varphi_{\mathcal{F}}, \varphi_{init})$, a finite trace is a *world trace* if it satisfies the fluent constraint, the action constraint, and the initial constraint φ_{init}. We further assume that the set $\mathcal{A}$ is partially ordered by a relation $\preceq$. This allows us to model related actions. For example, "pick an item" $\prec$ "pick two items".

In this work we consider task-like specifications: the ones where a concrete change in the environment needs to be accomplished in a one-off manner. We are not interested in infinite executions or specifications that can be satisfied by no action at all. Therefore, we assume that the user's specifications are co-safe LTL formulas, as they correspond to the described notion of task. Furthermore, we assume that the user is able to demonstrate the intended specification in a given world by a world trace.

4 INTERACTIVE SPECIFICATION SYNTHESIS

We now describe our algorithm for inferring an LTL specification starting with a natural language utterance and an example trace, and going interactively through a few distinguishing traces.

Figure 4 shows a high-level view of the interactive synthesis method. The method takes a world trace t and a natural language description d as an input (provided by a user). First, the trace t is used to create a sample $\mathcal{S}$ of positive and negative traces. Positive traces satisfy the specification, negative traces do not. From the information present in $\mathcal{S}$ and d, a set of candidate specifications C is generated. In order to determine which one of the candidates is the right one, LtlTalk generates

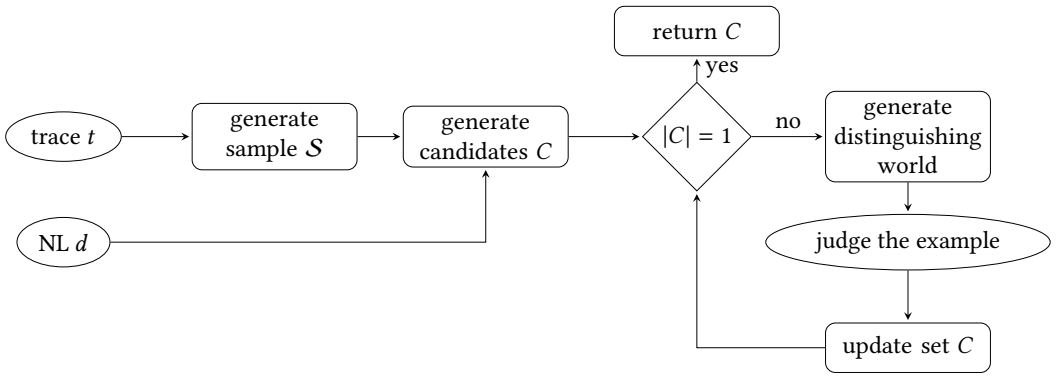


Fig. 4. High-level interactive synthesis algorithm

an initial world state and a world trace that are able to distinguish between the two most likely candidates. The user judges the provided example (by answering if the example matches the intent expressed in d). The user's verdict is then used to update the set of candidate specifications until only one candidate remains.

4.1 Constructing a Sample from a Single Example

The example provided by the user represents a positive example, a member of $\mathcal{P}$, the one that should be satisfied by the prospective specification. With only this one example, a space of potential specifications is too unconstrained, e.g., $\top$ would be a legitimate candidate specification, as it is consistent with the example. To shrink that space, we would like to infer a *sample* $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ that also contains negative examples. In order to infer what the set $\mathcal{N}$ might look like, we use the two heuristic principles: the *principle of no excessive trace* and the *principle of no excessive effort*.

The first principle says that if a prefix of the trace t would already be a good example for the user's intention, then the user would never bother to give the full trace t. Therefore, we add all the proper prefixes of the trace t to $\mathcal{N}$.

The second principle uses the partial order $\prec$, to express that the user will not demonstrate unnecessarily complex actions. The partial order in our robotic domain is naturally defined for picking actions: each picking action is a complex action consisting of multiple atomic picks of individual items. Thus, the partial order is defined by a subset relation between the picked items. Following the principle, any trace $t[a/a']$ that replaces an action a with a strictly lower action a' (i.e., $a' \prec a$) is added to $\mathcal{N}$.

Note that neither of the principles assumes that the user knows how to show a demonstration in a way that is optimal in any sense. Instead, they only assume that the user will not go beyond an already provided demonstration that is consistent with the specification.

For the user's example shown in Figure 2(a), the robot moving any part of the path from $(2, 4)$ to $(7, 4)$ constitutes a negative example, according to the principle of no excessive trace. The principle of no excessive action does not apply to this example, since the only picking action in the example is already a primitive action (picking an individual item).

4.2 Constructing a Set of Candidate Specifications

A formula ψ *perfectly separates* two sets of world traces, $\mathcal{P}$ and $\mathcal{N}$, if every trace from $\mathcal{P}$ satisfies ψ and every trace from $\mathcal{N}$ does not satisfy ψ. A formula that perfectly separates $\mathcal{P}$ from $\mathcal{N}$ is *consistent* with the sample $\mathcal{S}$.

Having created the sample $\mathcal{S}$, we are interested in finding a set of candidate specifications that are all consistent with $\mathcal{S}$. Moreover, we want to bias the search for candidate specifications using the natural language description d. The problem we solve is the following: given a world w, a sample $\mathcal{S}$, and natural language description d, create a set of candidate specifications C such that

a) the set of candidates C is consistent with the sample $\mathcal{S}$
b) $\psi \in C$ maximizes $\lambda(\psi, d)$, an idealized cost function capturing the connection between the specification and the natural language description.

Encoding Samples in SMT. As a first step, for a given sample $\mathcal{S}$ and integer hyperparameters δ and n, we show how finding a set of n formulas of size up to δ which are all consistent with $\mathcal{S}$ can be encoded as an SMT problem. Our encoding is similar to the encoding by Neider and Gavran [2018], where the authors show a SAT encoding of an LTL formula of a fixed size, consistent with a sample.

The complete encoding has three parts: the first part ensures that the syntax of the DAG is consistent with the LTL formula it represents, the second part ensures that every $t \in \mathcal{P}$ is satisfied by the DAG, and the third part ensures that every $t \in \mathcal{N}$ is not satisfied by the DAG. We call the resulting formula $\Phi_\delta^{\mathcal{S}}$. Full technical details of the construction can be seen in Appendix A.

The encoding captures the correspondence between a syntax DAG of size up to δ and an LTL formula that it represents: variables are representing nodes of the DAG and constraints are encoding the structure of the DAG. With a unique identifier i assigned to each node of the DAG, a variable $x_{i,o}$, for $o \in O$, is defined to track if a node i of the DAG is labelled by an operator o. Additional constraints connect a node of the DAG to its children, maintaining consistency with the arity of the labeling operator. That is, nodes with binary operations have two children, unary ones have a single child, and nullary operators have no children. We ensure that every node other than the root node has a parent (the DAG is connected).

The second set of constraints encodes the semantics of the syntax DAG on a trace. Given a trace t, we introduce variables $y^t_{i,\tau}$ to track if the subformula corresponding to the DAG node i evaluates to true at time step τ of the trace t. The encoding of the semantics follows the semantics of LTL. For example, if node i is labeled with X and its unique child is node j, then $y^t_{i,\tau}$ is true iff $y^t_{j,\tau+1}$ is true, if $\tau + 1 \leq |t|$. Other rules are similar.

Finally, ensuring consistency with the sample $(\mathcal{P}, \mathcal{N})$ amounts to requiring that for every trace $t \in \mathcal{P}$ it holds $y^t_{\Delta,0}$, with Δ being the identifier of the root node. Similarly, we require that for every trace $t' \in \mathcal{N}$ it holds $\neg y^{t'}_{\Delta,0}$

Optimization Modulo Theories. For our synthesis procedure, we require the candidate formulas to not only satisfy the constraints imposed by $\Phi_\delta^{\mathcal{S}}$, but also to find candidates that *optimize* certain cost functions. For this, we use optimization modulo theories. Recall that an *optimization modulo theories* problem [Bjørner et al. 2015] consists of the following components: a set of hard constraints in a background logic (containing Boolean satisfiability but also other theories); a set of soft constraints, each with a cost; and a set of cost functions. In any model, the hard constraints must be met. The soft constraints induce an additional cost function: each constraint may or may not be met, and the cost function adds up the weights of all constraints that are not met. The goal is to find a model that satisfies the hard constraints, and is optimal w.r.t. all the cost functions. Since there are multiple cost functions, an optimal solution is chosen from the Pareto front of solutions. A model is on the Pareto front if there is no model that performs better along all cost functions.

Natural Language Cost Function. Our cost function comes from the natural language description. Because the function λ is an idealized function, and is not readily available, we have to approximate it. Provided with a rich dataset, one could learn its approximation, as was done by Beltagy and

Quirk [2016]. Another option is using natural language processing tools to recover the syntactic structure of the description, and connect it to the structure of the formula, as done by Lignos et al. [2015].

Rich datasets are not commonly available and mapping linguistic structure to a formula is not helpful if there is only a weak signal present in the natural language description. Therefore, in this work we approximate $\lambda(\psi, d)$ by

$$L(\psi, d) := \sum_{\psi' \in subf(\psi)} h(lab(\psi'), d),$$

where the label of a formula, $lab(\psi) \in O$, is the top operator of the formula.

The approximation L uses a simple *hints* function h which assigns a score to each label of a subformula based on the natural language description d. Our synthesis method considers function h to be a black box that could be implemented in different ways. In our implementation of the robot simulation world, we define h by $h(o, d) = \frac{overlaps(o,d)}{|o|+|d|}$, which intuitively measures how much overlap there is between the natural language description and the operators and variables of the formula. We are using WordNet [Miller 1995] lemmas and their synonyms to compute h. Intuitively, for the natural language command d = *take one red item from 7,4* from the motivating example, a propositional variable p referring to red items will have a higher value $h(p, d)$ than the one referring to e.g. blue items.

We add two classes of soft constraints to the hard constraints captured by the formula Φ_δ^S. First, for all $o \in O$, we add a soft constraint $\bigvee_{i \leq \delta} x_{i,o}$ with a weight $h(o, d)$. This constraint suggests that the operator o should occur somewhere in the formula (hence, disjunction) and the strength of the suggestion is $h(o, d)$. Second, in order to avoid maximizing L by adding as many operators as possible, we additionally prefer smaller formulas. We define the integer variable Δ that captures the size of the found formula and add an objective to minimize the value of Δ. Finally, we combine the multiple objectives using the Pareto front strategy. The updated formula is named $\Phi_\delta^{S,L}$.

Solving the Constraint. With this encoding, we query an optimizing modulo theories solver to give us n models for $\Phi_\delta^{S,L}$, from which we extract the candidate specifications, each of maximum size δ. We iteratively query the solver for a solution, blocking the returned solutions before the next iteration. Note that blocking is syntactic: it is possible that two different, but semantically equivalent formulas could be found (e.g., $\mathsf{F}\, p \,\mathsf{U}\, p$ and $\mathsf{F}\, p$). The candidates are ranked implicitly by the order in which they are found (our optimization constraints prefer candidates that match the natural language description better). However, in order to determine the correct specification from the set of candidates, we have to interact with the user, as described next.

4.3 Generating Distinguishing World Traces

In order to narrow down the set of candidates C to a single candidate, LtlTalk iteratively offers (visual) examples consisting of an initial world and a trace in it for the user to judge. The world and the trace are generated in such a way that the user's verdict eliminates some of the candidates from C.

Given the two best ranked candidates, ψ_1 and ψ_2, fluents $\mathcal{F}$, actions $\mathcal{A}$ and the constraints $\varphi_\mathcal{F}$ and $\varphi_\mathcal{A}$, we aim to find an initial world φ_{init} and the world trace that satisfies one constraint but not the other, i.e., $(\psi_1 \wedge \neg\psi_2) \vee (\neg\psi_1 \wedge \psi_2)$. Simply finding a satisfying trace can be done in a standard way, e.g. by a language non-emptiness check for the intersection automaton, or by iterative SAT solving [Biere et al. 2006]. However, in this case, we need to pay attention to two more conditions:

- the found trace must be a world trace, i.e., it has to satisfy the fluent and action constraints $\varphi_\mathcal{F}$ and $\varphi_\mathcal{A}$,

```
Spec ⟶ RobotState | Spec or Spec | Spec and Spec | not Spec |  Spec until Spec | eventually Spec | Spec before Spec |
       Spec; Spec
RobotState ⟶ pick QItm at Loc | robot at Loc
Loc  ⟶ (Num, Num) | dry | wall | kitchen | bathroom | living room // Location attributes

// Item attributes:
QItm  ⟶ Quant FItm
Quant ⟶ Univ | Num
Univ  ⟶ every | all
FItm  ⟶ item | Fltr item
Fltr  ⟶ Prop | Fltr and Fltr |
Prop  ⟶  Color |  Shape
Color ⟶ red | blue | green | yellow | ...
Shape ⟶ triangle | square | circle | ...
```

Fig. 5. Core language syntax. Reserved constants and variable names are in italic.

- in order to make the user's decision about the shown trace effortless, the created world must be simple and the trace short.

We formalize these requirements as another multi-objective optimization modulo linear arithmetic problem and encode it in an optimizing SMT solver. Assuming the SAT formula that encodes a satisfaction of the difference specification for the trace of length ℓ_t, we add the following constraints and objectives to it. First, we add the constraints $\varphi_{\mathcal{A}}$ and $\varphi_{\mathcal{F}}$. Then, in order to keep the traces short, we add the objective to minimize ℓ_t. Finally, we define an integer variable c that captures the complexity of the world by counting the number of fluents that are true in φ_{init} and we require this variable to be minimized. A model for the final formula gives us the world and the trace that will distinguish between the two specification candidates.

5 GRAMMAR-BASED GENERALIZATION OF LEARNT SPECIFICATIONS

Natural language commands provided by the user serve two purposes. They are used to prune the candidate specifications in Section 4, but also to expand the lexicon of the system for later use.

The interactive specification synthesis technique from the previous section produces as output a mapping from a natural language utterance to an LTL specification. However, such a mapping is not robust to parameterization. For example, if a user has run the synthesis procedure to teach the system *pick one red from (7,4)*, it is unreasonable that the system requires them to run the synthesis procedure once again if they wish to *pick one triangle from (6,3)*. In this section, we describe a grammar-based generalization procedure which complements the LTL synthesis procedure from Section 4 by synthesizing parameterized specifications, thus improving the overall usability of the system.

We first describe our domain-specific core language that maps unambiguously to LTL, and which is the starting point of the grammar expansion procedure that we explain next.

5.1 The Core Specification Language

The grammar expansion starts with a domain-specific *core* language for specifying tasks in the blocks world. We call it a core language to emphasize that it will be expanded through the interaction with users. The grammar of the core specification language is shown in Figure 5. The basic actions (moving and picking) can be combined using Boolean and temporal connectives.

Note that LTL is contained in the core language. The grammar category `RobotState` corresponds to propositional variables. The keywords **`eventually`**, **`until`**, and **`before`** correspond to the temporal operators, and **`or`** and **`not`** to Boolean operators. We use the semi-colon (**;**) to denote chaining of two specifications.

The reason we need a core language is to distinguish the different concepts in the blocks world (positions, items, color, shape, etc.) into separate lexical categories. This enables the grammar expansion procedure to generalize a natural language utterance and induce new production rules. At the level of LTL, the categories are all encoded with propositional variables and syntactic generalization is not possible.

5.2 Grammar Expansion

Through interaction with users, the core grammar can be expanded using the *naturalization* process [Wang et al. 2017c]. Naturalization takes a pair of natural language description and the corresponding formal specification as its input and produces a grammar rule that generalizes from that pair. In LtlTalk, the input for naturalization comes either from the process of interactive synthesis (Section 4) or by the user giving a different name to an already successfully used specification (or a chain of specifications).

The task of expanding the grammar starts with a natural language description d, the corresponding formal specification s, and the model p. While s must be fully parsable using the current grammar's production rules, only some parts of d may be parsable. Using this vocabulary, for the example from Section 2, d = *take one red item from 7,4* and s = **`eventually`** `pick` *`one red`* `item at (7,4)`. In order to induce new rules, the system identifies *matches*—parsable spans appearing in both d and s. In our example, those are *`red`* (category `Prop`), *`one`* (category `Quant`), *`one red`* `item` (category `QItm`), `item` (category `FItm`), and (7, 4) (category `Loc`). A set of non-overlapping matches is called a *packing*, and is the basis for generating new grammar rules. Examples of packing are {*`red`*, `item` } or {*`one`*}, whereas { *`one red`* `item`, *`one`*} is not a packing because its elements are overlapping.

New grammar rules are introduced through *simple packing*, *best packing*, and *alignment*. (There can be more than one rule introduced per one description-specification pair.) Simple packing considers pre-defined primitive categories for matching (such as colors, shapes, or numbers). The primitive matches in the above example are numbers one, 7, and 4 (category `Num`), and color red (category `Color`). Therefore, a new rule is added to the grammar:

```
Spec ⟶ take Num Color item from (Num, Num) ≡ eventually pick Num Color item from Num Num
```

We use the symbol ≡ to connect the right-hand side of the induced rule to the core-language expression with the same semantics. From now on, LtlTalk will understand commands such as *take 2 yellow item from (1,3)*.

Best packing considers maximal packings, i.e., those that would become overlapping by adding any other match, and chooses the packing that scores the best under the model p. The best scoring maximal packing for our example results in the rule

```
Stmt ⟶ take QItm from Loc ≡ eventually pick QItm from Loc.
```

Note that this rule is more general than the one generated from the simple packing; with this rule in the grammar, LtlTalk will in the future understand commands such as *take every triangle item from kitchen*. However, it is not the case that best packing is a better method than simple packing: because of its eagerness to generalize as much as possible, it sometimes produces non-desirable new rules.

The third method, alignment, is applicable when the language description d and the specification s are almost identical: the non-identical parts are mapped to each other as synonyms. For more details on grammar learning, we refer readers to the work of Wang et al. [2017c].

The added production rules become first-class citizens of the grammar and can be combined with the core grammar rules freely. This enables LtlTalk to learn complex tasks through a combination of grammar-based naturalization and interactive synthesis. For instance, the command *take every triangle item from (4,3) or eventually robot at dry* is parsable immediately after the interactive synthesis of our working example.

LtlTalk can tolerate a number of wrong production rules being introduced in the process of grammar expansion (either by a wrong generalization or by a user's mistake). The model p assigns a score to each derivation. Every time LtlTalk parses a user's command, it offers a ranked list of candidate executions for the user to pick from. The user's choice is then used to update the model p. Thus, an undesirable production rule will be voted down and in effect excluded from the grammar.

6 EVALUATION

The LtlTalk implementation consists of three independent modules:

- a front-end module, which shows the simulated world to users and enables them to give examples for their commands (written in JavaScript);
- an interactive synthesis module, which implements the algorithms for synthesizing specifications from a natural language description and a single example, and for generating distinguishing examples, as described in Section 4 (written in Python). This module uses Z3 [De Moura and Bjørner 2008], an SMT solver that supports optimization modulo theories solving;
- an expanding formal language module, described in Section 5, implemented on top of the interactive version of the SEMPRE semantic parser toolkit [Wang et al. 2017c] (written in Java). This module additionally uses a thin NLP layer for lemmatization and number normalization [Manning et al. 2014].

In this section, we first evaluate the interactive synthesis algorithm in terms of performance and its ability to recover the intended specification. Then, we demonstrate how the grammar-based generalization complements interactive synthesis and allows LtlTalk to scale further.

6.1 Experimental Setup

We run all the experiments on a machine with four Intel Core i5-4590 CPUs at 3.3 GHz with 15 GB of RAM.

LTL Fragment. The exact operators and the set of propositional variables are relevant for the performance of the interactive synthesis algorithm. Ideally, one should use a language that is expressive enough to cover all interesting robot tasks in the blocks world, but is as small as possible. We explicitly use derived LTL operators, as using them makes specifications more compact. Concretely, alongside operators of the propositional logic, we use temporal operators $\{\mathsf{F}, \mathsf{U}, \mathsf{B}\}$. For a world of size 14×10, propositional variables are defined to determine the robot's location, the quality of robot's location (whether it is dry or wet), and the quantitative and qualitative properties of the picked items (what combination of color and shape properties, how many items, in numerical and *some* vs. *all* terms). We forbid nesting of multiple temporal operators beyond the B operator, whose definition involves nested temporal operators.

Benchmarks. We created 10 tasks, shown in Table 1. Each task consists of a natural language description, a world description and one example trace that are given to LtlTalk, as well as the

Table 1. Natural language descriptions and target specifications used for evaluation

task id	natural language description	specification	spec. size
t1	*step into water and then visit* $(6, 4)$	$\neg at(dry)$ B $at(6, 1)$	4
t2	*reach (4,1), but remain dry in the process*	$at(dry)$ U $at((4, 1)$	3
t3	*bring one green circle from* $(7, 4)$ *to* $(3, 4)$	$picked(1, green, circle, (7, 4))$ B $at(3, 4)$	3
t4	*take all green from* $(7, 4)$ *to water*	$picked(every, green, *, (7, 4))$ B $\neg at(dry)$	4
t5	*pick two square items from* $(4, 0)$	F$(picked(2, *, square, (4, 0))$	2
t6	*get one triangle from* $(4, 0)$ *and then one item of any kind from* $(11, 1)$	$picked(1, *, triangle, (4, 0)$ B $picked(1, *, *, (11, 1))$	3
t7	*get one item from* $(1, 2)$ *and one from* $(3, 1)$	F$(picked(1, *, *, (1, 2)))$ $\wedge$ F$(picked(1, *, *, (3, 1))$	5
t8	*get one green and one blue item from* $(7, 4)$	F$(picked(1, green, *, (7, 4)))$ $\wedge$ F$(picked(1, blue, *, (7, 4))$	5
t9	*reach* $(5, 4)$ *by only going through the water*	$\neg at(dry)$ U $at(5, 4)$	4
t10	*first get one red item from* $(7, 4)$ *and afterwards one green item from* $(10, 8)$	$picked(1, red, *, (7, 4))$ B $picked(1, green, *, (10, 8))$	3

Table 2. Performance of interactive synthesis for maximum size $\delta = 4$ and $n = 5$ initial candidates

task id	formula found	overall waiting time [s]	number of interactions
t1	yes (4/5)	4.5	2.8
t2	yes (5/5)	3.6	1.4
t3	yes (5/5)	6.28	2.6
t4	yes (4/5)	12.7	3.4
t5	yes (5/5)	14.4	2.4
t6	yes (5/5)	13.7	2
t7	no (0/5)	6.5	3
t8	no (0/5)	19.1	1.8
t9	yes (5/5)	3.2	2
t10	yes (5/5)	12.0	2.2

target specification, which is used to verify whether LtlTalk succeeded in synthesizing it. The specifications differ in their size (the size of their syntax DAG), ranging from two to five. As discussed in Section 5, these formulas should be viewed as language building blocks that can be combined in more complex formulas using naturalization.

6.2 Interactive Synthesis Evaluation

We assess the different parts of the proposed interactive synthesis algorithm in detail. In particular, we are interested in the following research questions:

RQ1 Does the algorithm synthesize specifications with only a few interaction rounds?
RQ2 Do natural language description and user interaction contribute to successful synthesis?
RQ3 How sensitive is the synthesis algorithm to parameter choices?
RQ4 How does our synthesis algorithm compare to enumerative synthesis?

RQ1: Ability to Synthesize Specifications. We run our synthesis algorithm on the ten tasks in Table 1, setting the maximum depth of the synthesized formula to $\delta = 4$ and the number of initial candidates generated to $n = 5$. Table 2 summarizes the results of this experiment. Because different runs of the optimizing solver may produce different models, we execute our algorithm five times per task and report how often out of the five runs the target specification was found, as well as the average number of interaction rounds.

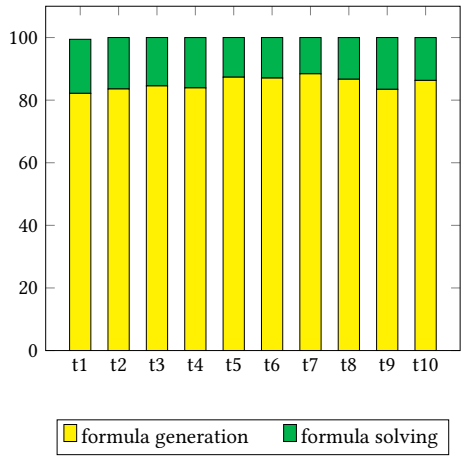


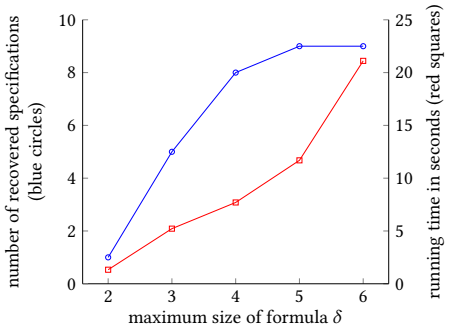
Fig. 6. Relative times needed to generate and to solve formula $\Phi_{\delta}^{S,L}$

In the majority of cases, LtlTalk found the target formula. It failed to do so for the tasks t7 and t8, for which the target formula has size 5, which is beyond our default limit of $\delta = 4$. The average number of interactions was 2.4, which we consider to be small enough to not overly burden the user.

We measure the user's overall waiting time—the time spent waiting for the system to devise initial candidates as well as to devise distinguishing examples. The average overall waiting time for the user is 9.6 seconds. We note that most of this waiting time is spent on creating the initial candidate set, and in particular on generating the propositional formula $\Phi_{\delta}^{S,L}$. Figure 6 shows the relative times needed to generate $\Phi_{\delta}^{S,L}$ compared to the time needed to solve it (i.e. to find the satisfying assignment). We observe that it takes much longer to generate a formula than to solve it. The average times needed for solving formulas from Table 2 and for generating distinguishing examples are both below one second. While we do not consider the waiting time to be prohibitively long, we note that a more optimized implementation of the formula generation (e.g. in C++ instead of in Python) would likely reduce the overall running time of our synthesis.

RQ2: Role of NL and User Interaction. We further examine the role that the natural language hints play. We ran the same experiment, but supplied no hints information when generating the formula $\Phi_{\delta}^{S,L}$. Without the hints, the intended formula was recovered in only 9 out of 50 cases. Clearly, increasing the size of the initial candidates set n would allow the algorithm to recover the formula without the natural language hints eventually. This would, however, increase the candidate generation time and the number of interactions needed with the user, impacting the user experience.

We next evaluate the role interaction plays in the overall synthesis procedure. In particular, we evaluate whether the initial candidates generation is already enough, i.e., whether it could stand on its own, without additional interaction with the user. Among the 50 runs, the target specification was the top-ranked candidate in 15 cases, it was among the top 3 candidates in 34 cases, and among the top 5 in 38 cases. With non-expert users in mind, we consider only the top-ranked candidate as a successful trial. This means that without interaction, in only 15 experiments the specification would be successfully recovered. We conclude that the interaction is an important part of the



(a) Varying δ, with n set to 5

(b) Varying n, with δ set to 4

Fig. 7. Number of recovered specifications and running time for different choices of hyperparameters n and δ

synthesis procedure, as it manages to narrow down the set of candidate specifications to a single, correct specification.

RQ3: Algorithm Parameter Sensitivity. Our synthesis algorithm has two hyperparameters: the maximum size of the specification δ and the number of initial candidates n. The larger these two parameters are, the longer the synthesis runs, but also the more tasks can be successfully solved. We conducted a number of preliminary experiments, varying δ and n. Figure 7 shows the number of successfully recovered specifications and the running time for different values of δ (Figure 7(a)) and for different values of n (Figure 7(b)).

For $\delta \geq 5$ there is a possibility to recover all the specifications from benchmarks, since the target specifications do not exceed this depth. However, for $\delta = 5$ or $\delta = 6$, the algorithm successfully recovered only 9 out of 10 specifications. The task t7 (which has size 5) was not recovered, even though the similar task t8 was. A closer inspection reveals that the synthesized specification for t7 was $\mathit{picked}(1, *, *, (1, 2))$ B $\mathit{picked}(1, *, *, (3, 1))$. This shorter specification was consistent with a single example (as were the other candidate specifications), and the natural language description was not able to bias the search towards the correct specification. A different example trace or a better NL similarity function would be necessary to recover the intended specification.

So far, we have insisted on the user providing only a single example. However, depending on the application, it may be the case that asking for few examples would not harm the user experience. To see what the effects of going beyond a single example are, we re-ran the interactive synthesis on the same benchmark set, but with two examples provided for each specification instead of just one. The pre-interaction results improve: the target specification was the top-ranked candidate in 22 cases, it was in the top 3 in 35 cases and in top 5 in 39 cases. Additionally, the specification for the task $t7$ is now successfully recovered with $\delta = 5$. Thus, by using more initial examples, one might choose a smaller number of initial candidates, at the expense of higher user effort.

As described in subsection 4.1, our algorithm relies on two principles for devising a set of negative examples: the principle of no excessive trace and the principle of no excessive effort. We evaluated what the relative contributions of those principles to the performance of the algorithm are by running the same experiment, but giving up on using one of the principles at a time. When negative examples were generated without the no excessive trace principle, the results were significantly worse (as can be seen in Table 3). When the principle of no excessive effort was not used, the system was not able to recover $t8$ for the specification depth $\delta = 5$ (which it was able to do using

Table 3. Performance of interactive synthesis for maximum size $\delta = 4$ and $n = 5$ initial candidates *without using the no excessive trace principle for generation of negative examples*

task id	formula found	overall waiting time [s]	number of interactions
t1	no (2/5)	3.5	2.8
t2	no (2/5)	2.7	2.6
t3	no (1/5)	2.1	3.4
t4	no (0/5)	4.0	3.0
t5	yes (5/5)	2.3	2.6
t6	no (1/5)	3.9	2.4
t7	no (0/5)	3.5	2.2
t8	no (0/5)	2.5	2
t9	no (0/5)	3.7	3.2
t10	yes (5/5)	4.6	3.2

the principle). On the other hand, not using the no excessive trace principle results in reducing overall waiting time for the user to below 10 seconds for each of the tasks.

Finally, we performed an additional experiment, whereby we encoded the generation of distinguishing worlds eagerly, already in the formula $\Phi_\delta^{\mathcal{S},L}$. This has potential advantages over the current, lazy generation, approach: no waiting time for the user between interactions, never finding two semantically equivalent specifications (for traces up to a fixed length), and a possibility to take a smarter strategy in choosing which disambiguating example to present the first. An obvious drawback is that eager generation increases the size of $\Phi_\delta^{\mathcal{S},L}$. Unfortunately, we found the eager generation approach to not scale well: the average waiting time for initial candidates rose to 70 seconds, with 10 solver timeouts (for a timeout set to 120 seconds). We thus stick to lazy generation.

RQ4: Comparison with Enumerative Synthesis. A natural baseline for generating initial candidates is enumerative synthesis [Alur et al. 2015]. We ran two variants of the algorithm: enumerating expressions from the language in the order of size and enumerating expressions from the language biased by their similarity to the natural language description. We enumerated over the same language that was used in the previous experiments, in particular using only those propositional variables that appeared in the example trace. To compare with our approach, we let the enumeration run for 20 seconds (the total time our algorithm at most takes). We then check whether the target specification is contained in the set of specifications consistent with the examples (the user-provided positive and generated negative examples), i.e. whether the system could possibly find it through disambiguating interactions with the user.

The size-based enumeration successfully found four specifications: $t1$, $t2$, $t5$ and $t9$. While $t5$ is the smallest size specification, we observe that none of $t1$, $t2$ and $t9$ contains a picking action. If an example contains picking of items, there are many more propositional variables to consider, which means that the size of the language to enumerate increases, making the enumeration more difficult. For example, picking a red circle from $(1, 2)$ that contains only that item makes the following propositions true: $picked(1, red, *, (1, 2))$, $picked(1, *, circle, (1, 2))$, $picked(1, red, circle, (1, 2))$, $picked(every, red, *, (1, 2))$, $picked(every, *, circle, (1, 2))$, $picked(every, red, circle, (1, 2))$.

After adding the natural language bias, enumeration is able to successfully find five specifications: $t1$, $t2$, $t3$, $t9$ and $t10$. We conclude that our SMT-based method for deriving initial candidates is superior to enumeration.

Table 4. Example expressions expanding the core grammar

NL description	generated grammar rule	newly parsable commands
(t1) step into water and then visit (6, 4)		
	`step into water and then visit (Num, Num)`	step into water and then visit (1, 1)
	`step into water and then visit Loc`	step into water and then visit kitchen
		...
(t2) reach (4,1), but remain dry in the process		
	`reach (Num, Num) but remain` *`dry`* `in the process`	reach (10,8) but remain dry in the process
	`reach Loc but remain Loc in the process`	...
(t3) bring one green circle from (7, 4) to (5, 4)		
	`bring` *`one`* `Color Shape from (Num, Num) to (Num, Num)`	bring every blue square from kitchen to bathroom
	`bring Quant Prop Prop from Loc to Loc`	...
(t4) take all green from (7, 4) to water		
	`take` *`every`* `color from (Num, Num) to water`	take two squares from (1,3) to water
	`take Quant Prop from Loc to water`	take one item at (2,4)
	`take` ≡`pick`	...
(t10) first get one red item from (7, 4) and afterwards one green item from (10, 8)		
	`first get Quant Prop item from Loc`	first get all triangle items from (2,1)
	`and afterwards Quant Prop item from Loc`	and afterwards two red items from kitchen
	`from` ≡`at`	take one item from kitchen
	...	...

6.3 Case Study for Grammar-Based Generalization

We illustrate our grammar-based generalization from Section 5 using case studies. First, we consider the examples from subsection 6.2 and show that LtlTalk learns not only the individual demonstrated tasks (t1 - t10), but a whole class of tasks obtained by generalization. Then we show how grammar-based generalization can be used for providing complex but repetitive specifications in an easy way. Finally, we show how the generalization helps with tasks from robotic dialog systems literature.

Generalization of Interactive Synthesis Tasks. Table 4 shows induced specification expressions alongside examples of newly parsable commands for a subset of examples from Table 1. When a derived production rule does not have `Spec` as its left-hand side, we add to the table a core-language expression with which it shares the semantics, and denote this by the symbol ≡.

Let us first consider the task t4. Its natural language description is *take all green from* (7, 4) *to water* and the learnt core language specification is `pick` *`every green`* `item at (7,4)` **`before not`** *`robot`* `at` *`dry`*. After generalization, the command *take two squares from (1,3) to water* is immediately understood. Namely, using best packing method, the expression `take Quant Prop from Loc to water` is added to the language. Using the alignment method, LtlTalk furthermore learnt that *take* can be used interchangeably with `pick`, and thus an expressions such as *take one item at (2,4)* is now a part of the language.

Note that the system has not generalized over the word *water*. The reason is that *water* is not even partially parsable—it does not appear in the grammar (and the properties of wet tiles can only be expressed by negating the grammatical entity *dry*). Hence, which generalizations are produced also depends on how the grammar is designed.

The generalization from task t3 enables the system to understand expressions such as *bring every blue square from kitchen to bathroom*. However, the same generalization allows for some nonsensical expressions to become parsable: e.g., *bring three yellow red from kitchen to bathroom*.

```
pick every red item at (2,3) before robot at (1,1)
and
pick every blue item at (2,3) before robot at (1,10)
and
pick every green item at (2,3) before robot at (14,1)
and
pick every yellow item at (2,3) before robot at (14,10)
```

Fig. 8. Sorting items based on colors using the core language

As discussed in Section 5, this is a property of the best packing technique: it generalizes aggressively, at a price of introducing spurious expressions to the language.

Complex Specifications. The naturalization technique proves to be especially useful for scenarios in which the robot does many structurally similar tasks over and over again, e.g., a hospital robot taking different drugs to different patients. In our abstract blocks world, consider a task of distributing items of different color from a selected location (for instance, (2,3)) to the four corners of the world. Conceptually, the task is very simple: the robot first needs to take all red items to the lower-left corner, then it needs to take all blue items to the upper-left corner, then it needs to take all green items to the lower-right corner, and finally it needs to take all yellow items to the upper-right corner. Its specification in the core language, shown in Figure 8, is complicated and repetitive. Furthermore, the specification is of depth 15. Our interactive synthesis algorithm is not able to uncover such a long specification.

The solution comes from the modularity of the target specification. Using interactive synthesis, the user can first specify the command *all red from 2,3 to 1,1*. LtlTalk generalizes from the specification and now knows the meaning of any command that corresponds to `Quant Prop from Loc to Loc`. Now the specification from Figure 8 can be written by using the conceptual idea of the task, saying *all red from 2,3 to 1,1 and all blue from 2,3 to 1,10 and all green from 2,3 to 15,1 and all yellow from 2,3 to 14,10* (while this specification is also fairly long, it assumes much less knowledge about the core language: only that individual specifications can be connected by the operator `and`).

Finally, this specification can be renamed into *distribute colors from 2,3*. In the future, then, the user could do the same for any other location using a single command.

Dialog Systems Tasks. Now we turn our attention to tasks inspired by three studies of robotic dialog systems [Kollar et al. 2013; Perera and Veloso 2015; Thomason et al. 2015]. The idea of a dialog system is that the robot can ask its user clarifying questions to understand the task. The tasks are of two kinds: *navigation* and *delivery*. Here, we show how LtlTalk is able to do typical navigation and delivery tasks, but also go beyond those.

The tasks of interest are navigation (e.g., *go to the bathroom*) [Kollar et al. 2013; Thomason et al. 2015], delivery (e.g., *bring a spoon from the living room to the kitchen*) [Thomason et al. 2015], and a complex combinations of atomic tasks (e.g., *go to the bathroom and then go to the living room*) [Perera and Veloso 2015].

Assume first that a user of LtlTalk tasks the robot to *eventually be at the restroom*. After the user provides the example (and potentially judges the examples provided by LtlTalk), the system learns that *restroom* is the same as *bathroom* (using *alignment*). Similarly, after the user demonstrates the command *go to the kitchen*, LtlTalk knows that the meaning of this phrase corresponds to `eventually at the kitchen` (using *simple packing*).

It is not only learning of a new phrase that happened, but also generalization. To illustrate, the expression *go to the restroom* is now a part of the language of LtlTalk, as it learnt the meaning of `go to Loc` as well as the synonymity of *bathroom* and *restroom*.

The generalization happens over different grammar categories. For instance, assuming that the user demonstrates the command *go to the kitchen and then go to the living room*, LtlTalk will introduce a new rule to its grammar, encoding that two categories `Spec` can be connected with the connector *and then*, thus forming a new production rule `Spec` $\longrightarrow$ `Spec and then Spec` $\equiv$ `Spec before Spec` (using *best packing*). This allows for generalizing to kinds of actions different than navigation only, e.g., *pick one red item at the kitchen and then go to the living room* is now a part of the LtlTalk's language.

These examples show how LtlTalk's approach can achieve the functionality of the existing dialog systems. These systems ask the user for particular parts of the action, which makes them dependent on knowing the exact structure of possible specifications. LtlTalk on the other hand, gets the clarification from the user by way of demonstration, therefore providing a more general solution.

7 RELATED WORK

We presented LtlTalk, a *natural language robotic interface* using interactive specification synthesis *from examples* and *natural language descriptions*. In all three areas (NL communication with robots, synthesis from natural language, and synthesis from examples) recent years have brought a lot of interesting developments. In this section we relate LtlTalk to the works in these topics.

Natural Language Interfaces for Robotics. In an attempt to provide a more natural specification language for robotics, but keep the precision of a formal language, Kress-Gazit et al. [2008] propose a controlled, natural looking language that matches a fragment of LTL. SLURP [Lignos et al. 2015] uses NLP techniques to map the linguistic structure of a command to a fragment of LTL. LtlTalk shares the usage of LTL as its underlying expressive and precise specification language, but adapts the formal language to the users' style through interaction with them.

The grammar expansion technique that we use originates from Voxelurn [Wang et al. 2017c], an NL instruction system for 3D-blocks building world. There, a user is expected to provide a formal specification for a natural language description that was not understood by the system. LtlTalk removes that burden from its users and enables them to provide an example instead (which is then turned into a formal specification using the interactive synthesis algorithm).

The early work in the robotic dialog systems [Kollar et al. 2013; Meriçli et al. 2014] puts forward the idea of understanding a specification through interactions with the user. They learn the new expressions, but do not generalize. Generalization of the learnt expressions is achieved in the work by Thomason et al. [2015] by using the induction of a CCG grammar from the semantic parsing framework SPF [Artzi 2016]. The commands are limited to delivery and navigation tasks, in contrast to LtlTalk's ability to handle temporal specifications with different propositional variables.

Synthesis from Natural Language. NLyze [Gulwani and Marron 2014] proposes a natural language interface to spreadsheet programming where a natural language utterance is mapped to a DSL using a semantic parser; the users resolve ambiguity by selecting the correct spreadsheet macro from a ranked list of candidates. SQLizer [Yaghmazadeh et al. 2017] proposes an NL interface for SQL

query programming. (The users are assumed to be unaware of the underlying tables' structure, so providing examples is not an option.) A semantic parser is used as a front-end to generate sketches of SQL queries and ambiguity is resolved by program repair. Similarly, NaLIR [Li and Jagadish 2014] uses an NL interface for SQL programming, and lets users select correct queries. Compared to these systems, LtlTalk *actively* asks for user input by showing new, disambiguating, worlds.

Many successful NL to DSL systems use large datasets of natural language descriptions and corresponding formal language commands to train the synthesizer's model [Balog et al. 2017; Beltagy and Quirk 2016; Desai et al. 2016; Polosukhin and Skidanov 2018]. Instead of requiring a large training set upfront, which is challenging to obtain, LtlTalk learns over time and adapts to idioms and user-specific ways of expressing commands.

Synthesis from Examples. Example-based synthesis techniques have developed several strategies to resolve the inherent ambiguity. Similarly to LtlTalk, Scythe [Wang et al. 2017a,b], a system for learning SQL queries from input-output examples, uses active querying for disambiguation. Beyond the different application domains, LtlTalk integrates the semantic parser more tightly, as the core language is gradually extended based on user interaction.

Several techniques have been developed to reduce user effort in example-based synthesis. SketchAX [An et al. 2020] leverages properties such as invariance to input perturbations to generate an additional set of examples. Drachsler-Cohen et al. [2017] propose a system that interacts with a user by generating abstract examples, which represent a set of concrete examples and thus reduces the rounds of interaction. FlashProg [Mayer et al. 2015] allows users to inspect generated programs in a compact form or asks clarifying questions based on existing test data. Peleg et al. [2018] develop a Granular Interaction Model, which on one hand shows evaluation results at intermediate steps in a program, and on the other hand asks users to inspect generated programs and to provide feedback on parts of it. These techniques do not directly apply to our domain of LTL specification. In particular, unlike the programs in the target domains of these works (integer and bitvector manipulating programs, string processing), LTL specifications are challenging to inspect by users.

Vazquez-Chanlatte et al. [2018] use the principle of maximum entropy to define the problem of learning a temporal specification from only positive examples. In order to solve the problem, they have to pre-compute a lattice of implication relations between the specifications.

Synthesis from Natural Language and Examples. Combining example-based specifications with natural language descriptions has also been explored previously to reduce the number of examples that a user has to provide. Common with LtlTalk, these techniques use the natural language description of a task to bias the search for the correct program towards more likely candidates, but other details differ which make them not immediately applicable to synthesizing LTL specifications. Manshadi et al. [2013] use a dependency parser to bias the search for regular expressions, but the underlying synthesis relies on version space algebra. Regel [Chen et al. 2020] uses a semantic parser to obtain the basic scaffolding for the target regular expression from the user's NL description, and then completes the expression using programming-by-example. Nye et al. [2019] use a neural network instead of a semantic parser to generate sketches that are filled using enumerative synthesis. Mars [Chen et al. 2019] encodes synthesis as a MAX-SMT problem, similar to LtlTalk, but trains a neural network to provide the weights. LtlTalk avoids the training phase by adapting over time using an extensible semantic parser. Finally, Raza et al. [2015] use a semantic parser to split the natural language description into smaller parts for which the user can separately provide examples. LtlTalk's generalization procedure similarly allows to combine smaller tasks into more complex ones, but the composition happens in the semantic parser itself and does not require re-synthesizing the low-level tasks for new combinations. Neither of the above mentioned techniques uses active disambiguation through user interaction.

Learning Automata and Temporal Logics. Learning regular languages from positive and negative examples is a classical problem in computational learning theory [Angluin 1987; Gold 1967]. Angluin's L^* algorithm, which shows polynomial time learning of regular languages using traces and an equivalence oracle [Angluin 1987] has since found many distinct applications in formal methods [Cassel et al. 2016; Cobleigh et al. 2003]. More recently, learning LTL formulas from traces was used in the context of specification mining [Camacho and McIlraith 2019; Kim et al. 2019; Neider and Gavran 2018], together with a rich body of work on mining STL specifications [Bartocci et al. 2013; Jin et al. 2015; Kong et al. 2014, 2017; Mohammadinejad et al. 2020]. These applications have not considered the "one-shot" setting, where the language must be learnt with a single example and few interactions. LtlTalk builds upon the encoding by Neider and Gavran [2018] to create an algorithm for such a setting. Incidentally, while a classical result shows the problem of learning small automata from samples is NP-hard, there is no analogous result known for LTL.

8 CONCLUSION

We have described a combination of example-driven synthesis and grammar-based naturalization that allows "one-shot" learning and generalization of LTL specifications from natural language utterances and examples.

Like all techniques based on programming by example, our techniques are ultimately incomplete. For example, it is possible that the candidate generation fails to find the right specification. This can happen if the hyperparameter n (number of initial candidates) is too small. Another example is if no distinguishing worlds are found. That can happen either because the candidate formulas are semantically equivalent or because the bound on lengths of world traces is too small. The result in both of those cases is that the grammar will contain a wrong production rule.

As shown in our experimental evaluation, such incompleteness is rare in practice. Moreover, the advantage of combining with naturalization is that the probabilistic grammar model is robust to a few wrong production rules (and can effectively eliminate the wrong rules through interaction with the user.)

The combination of programming by example and naturalization opens the door to more complex synthesis procedures. For example, it allows users to flexibly and programmatically combine natural language directives taught through examples and generalized by the system. We believe this combination has potential beyond the world of end-user programming for robotics applications.

Acknowledgements The authors want to thank Eman Eman and Chuntong Gao for their work on improving the LtlTalk interface. This research was partially funded by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248–CPEC (see https://perspicuous-computing.science) and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

A FINDING AN LTL FORMULA CONSISTENT WITH A SAMPLE

A variant of this problem of finding a smallest-size formula consistent with a sample $\mathcal{S}$, was previously solved by encoding it as a SAT problem [Neider and Gavran 2018]: iterating over consecutive values of $i \in \mathbb{N}$, the propositional formula $\Phi_i^{\mathcal{S}}$ is created that is satisfiable if and only if there is an LTL formula ψ consistent with the sample. We modify this encoding slightly: instead of iterating over the exact size i, the maximum size δ is added into the encoding. We review the modified encoding next.

Figure 9 shows the part of the encoding that captures the syntactic properties of a syntax DAG. Every node of the syntax DAG is assigned a unique identifier $i \in \mathbb{N}$. We define a variable $x_{i,o}$ to be true if and only if the operator o is at the node identified by i, for $i \leq \delta$. We define variables $l_{i,j}$ and $r_{i,j}$ to be true if and only if the left (right) child of node i is the node j. To determine the size of the DAG, variable d_i is true if and only if the DAG has at least i nodes.

Formula 1 in Figure 9 makes sure that the DAG is of the size at least 1 and that d_i implies d_j, $\forall j < i$. The auxiliary variable ρ_i is true if and only if the formula represented by the DAG is of size i. Formula 2 encodes that every node in the DAG corresponds to exactly one operator, and that no symbol is defined for a variable x_i with index i larger than the size of the DAG. Formula 3 and Formula 4 requires that every node has the correct number of children. That is, every unary (U) or binary (B) operator has exactly one left child (by convention, unary operators only have a left child), and propositional variables have no left children. Similarly, every binary operator has exactly one right child, and all other operators have no right children. Finally, Formula 5 says that the operator at location 1 is a propositional variable and that every node has a parent node (except for the root node). We denote the conjunction of all the formulas from Figure 9 by Φ^{DAG}.

In order to capture the semantic properties of the operators and connect them to the traces, variables $y_{i,\tau}^t$ are defined to be true if and only if a subformula corresponding to the node i evaluates to true at timestep value τ of trace t. For instance, the operator at node 4 being F and its left child being the node 2 ($x_{4,\mathsf{F}} = \top \wedge l_{4,2} = \top$), implies that $y_{4,\tau}^t$ is true if and only the disjunction $\bigvee_{\tau' \geq \tau} y_{2,\tau'}^t$ is true. For another example, the operator at node 1 being a propositional variable q, implies that $y_{1,\tau}^t$ is true if and only if $q \in t[\tau]$.

$$\left[\bigvee_{1\leq i\leq\delta} d_i \wedge \bigwedge_{2\leq i\leq\delta} d_i \rightarrow d_{i-1}\right] \wedge \left[\rho_\delta \leftrightarrow d_\delta \wedge \bigwedge_{i<\delta} \rho_i \leftrightarrow d_i \wedge \neg d_{i+1}\right] \tag{1}$$

$$\left[\bigwedge_{1\leq i\leq\delta} d_i \rightarrow \left(\bigvee_{o\in O} x_{i,o} \wedge \bigwedge_{o\neq o'\in O} \neg x_{i,o} \vee \neg x_{i,o'}\right)\right] \wedge \left[\bigwedge_{1\leq i\leq\delta} \neg d_i \rightarrow \bigwedge_{o\in O} \neg x_{i,o}\right] \tag{2}$$

$$\left[\bigwedge_{2\leq i\leq\delta} \bigwedge_{o\in U\cup B} x_{i,o} \rightarrow \left(\bigvee_{1\leq j<i} l_{i,j} \wedge \bigwedge_{1\leq j<j'<i} \neg l_{i,j} \vee \neg l_{i,j'}\right)\right] \wedge \left[\bigwedge_{2\leq i\leq\delta} \bigwedge_{o\in Q} x_{i,o} \rightarrow \bigwedge_{1\leq j<i} \neg l_{i,j}\right] \tag{3}$$

$$\left[\bigwedge_{2\leq i\leq\delta} \bigwedge_{o\in B} x_{i,o} \rightarrow \left(\bigvee_{1\leq j<i} r_{i,j} \wedge \bigwedge_{1\leq j<j'<i} \neg r_{i,j} \vee \neg r_{i,j'}\right)\right] \wedge \left[\bigwedge_{2\leq i\leq\delta} \bigwedge_{o\in Q\cup U} x_{i,o} \rightarrow \bigwedge_{1\leq j<i} \neg r_{i,j}\right] \tag{4}$$

$$\left[\bigvee_{q\in Q} x_{1,q}\right] \wedge \left[\bigwedge_{1\leq i<\delta} d_i \rightarrow \left(\bigvee_{i<j\leq\delta} d_j \wedge (l_{j,i} \vee r_{j,i})\right)\right] \tag{5}$$

Fig. 9. Encoding of syntactic properties of a syntax DAG

$$\bigwedge_{1\le i\le\delta}\bigwedge_{q\in Q} x_{i,q} \rightarrow \left[\bigwedge_{0\le\tau<|t|}\begin{cases} y^t_{i,\tau} & \text{if } q\in t(\tau)\\ \neg y^t_{i,\tau} & \text{if } q\notin t(\tau)\end{cases}\right] \tag{6}$$

$$\bigwedge_{\substack{1<i\le\delta\\1\le j<i}} (x_{i,\neg}\wedge l_{i,j}) \rightarrow \bigwedge_{0\le\tau<|t|}\left[y^t_{i,\tau}\leftrightarrow\neg y^t_{j,\tau}\right] \tag{7}$$

$$\bigwedge_{\substack{1<i\le\delta\\1\le j,j'<i}} (x_{i,\vee}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow \bigwedge_{0\le\tau<|t|}\left[y^t_{i,\tau}\leftrightarrow(y^t_{j,\tau}\vee y^t_{j',\tau})\right] \tag{8}$$

$$\bigwedge_{\substack{1<i\le\delta\\1\le j<i}} (x_{i,\mathsf{X}}\wedge l_{i,j}) \rightarrow \left[\bigwedge_{0\le\tau<|t|-1} y^t_{i,\tau}\leftrightarrow y^t_{i,\tau+1}\wedge\neg y^t_{i,|t|-1}\right] \tag{9}$$

$$\bigwedge_{\substack{1<i\le\delta\\1\le j,j'<i}} (x_{i,\mathsf{U}}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\le\tau<|t|} y^t_{i,\tau}\leftrightarrow\bigvee_{\tau\le\tau'<|t|}\left[y^t_{j',\tau'}\wedge\bigwedge_{\tau\le\tau''<\tau'} y^t_{j,\tau''}\right]\right] \tag{10}$$

Fig. 10. Constraints enforcing that the variables $y^t_{i,\tau}$ track the valuation of the prospective LTL formula for a fixed trace t

Figure 10 shows the encoding of the semantic properties of a syntax DAG. Formula 6 connects the values of propositional variables to the trace t. The following formulas encode the semantics of the operators $\neg$, $\vee$, X and U. (In a similar fashion, one encodes the semantic properties of any derived operators, such as F or G.) We denote the conjunction of all the formulas from Figure 10 with Φ_t.

Finally, the complete encoding requires that the formula is consistent with the sample:

$$\Phi^{\mathcal{S}}_{\delta} := \Phi^{\text{DAG}} \wedge \bigwedge_{t\in\mathcal{P}}\left[\Phi_t\wedge\bigwedge_{1\le i\le\delta}\rho_i\rightarrow y^t_{i,0}\right]\wedge\bigwedge_{t\in\mathcal{N}}\left[\Phi_t\wedge\bigwedge_{1\le i\le\delta}\rho_i\rightarrow\neg y^t_{i,0}\right]$$

REFERENCES

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2020. Augmented example-based synthesis using relational perturbation properties. *PACMPL* 4, POPL (2020). https://doi.org/10.1145/3371124

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987). https://doi.org/10.1016/0890-5401(87)90052-6

Yoav Artzi. 2016. Cornell SPF: Cornell Semantic Parsing Framework. arXiv:arXiv:1311.3011

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations (ICLR)*.

Ezio Bartocci, Luca Bortolussi, and Guido Sanguinetti. 2013. Learning Temporal Logical Properties Discriminating ECG models of Cardiac Arrhytmias. *CoRR* abs/1312.7523 (2013). arXiv:1312.7523

I. Beltagy and Chris Quirk. 2016. Improved Semantic Parsers For If-Then Statements. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. https://doi.org/10.18653/v1/P16-1069

Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. 2006. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science* 2, 5 (2006). https://doi.org/10.2168/LMCS-2(5:5)2006

Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-662-46681-0_14

Alberto Camacho and Sheila A. McIlraith. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal Asp. Comput.* 28, 2 (2016). https://doi.org/10.1007/s00165-016-0355-5

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal Synthesis of Regular Expressions. In *Programming Language Design and Implementation (PLDI)*.

Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-layer Specification Synthesis. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/3338906.3338951

Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/3-540-36577-X_24

Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Giuseppe De Giacomo and Moshe Y. Vardi. 2015. Synthesis for LTL and LDL on Finite Traces. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-540-78800-3_24

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884786

Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-63387-9_13

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/302405.302672

R. Fikes and N. J. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.* 2, 3/4 (1971). https://doi.org/10.1016/0004-3702(71)90010-5

Ivan Gavran, Rupak Majumdar, and Indranil Saha. 2017. Antlab: A Multi-Robot Task Server. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 190 (2017). https://doi.org/10.1145/3126513

M. Ghallab, C. Aeronautiques, C. K. Isi, and D. Wilkins. 1998. *PDDL: The Planning Domain Definition Language.* Technical Report CVC TR98003/DCS TR1165. Yale Center for Computational Vision and Control.

E. Mark Gold. 1967. Language Identification in the Limit. *Information and Control* 10, 5 (1967). https://doi.org/10.1016/S0019-9958(67)91165-5

Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/2588555.2612177

Gerard J. Holzmann. 2002. The logic of bugs. In *Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/587051.587064

Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. 2015. Mining Requirements From Closed-Loop Control Models. *IEEE Trans. on CAD of Integrated Circuits and Systems* 34, 11 (2015). https://doi.org/10.1109/TCAD.2015.2421907

Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. 2019. Bayesian Inference of Linear Temporal Logic Specifications for Contrastive Explanations. In *International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.24963/ijcai.2019/776

Thomas Kollar, Vittorio Perera, Daniele Nardi, and Manuela M. Veloso. 2013. Learning Environmental Knowledge from Task-Based Human-Robot Dialog. In *International Conference on Robotics and Automation (ICRA)*. https://doi.org/10.1109/ICRA.2013.6631186

Zhaodan Kong, Austin Jones, Ana Medina Ayala, Ebru Aydin Gol, and Calin Belta. 2014. Temporal logic inference for classification and prediction from data. In *Hybrid Systems: Computation and Control (HSCC)*. https://doi.org/10.1145/2562059.2562146

Zhaodan Kong, Austin Jones, and Calin Belta. 2017. Temporal Logics for Learning and Detection of Anomalous Behavior. *IEEE Trans. Automat. Contr.* 62, 3 (2017). https://doi.org/10.1109/TAC.2016.2585083

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2008. Translating Structured English to Robot Controllers. *Advanced Robotics* 22, 12 (2008). https://doi.org/10.1163/156855308X344864

H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics* (2009). https://doi.org/10.1109/TRO.2009.2030225

H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R.B. Scherl. 1997. GOLOG: A Logic Programming Language for Dynamic Domains. *J. Log. Program.* 31, 1-3 (1997). https://doi.org/10.1016/S0743-1066(96)00121-5

Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (2014). https://doi.org/10.14778/2735461.2735468

Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell P. Marcus, and Hadas Kress-Gazit. 2015. Provably Correct Reactive Control from Natural Language. *Auton. Robots* 38, 1 (2015). https://doi.org/10.1007/s10514-014-9418-8

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. https://doi.org/10.3115/v1/P14-5010

Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. 2013. Integrating Programming by Example and Natural Language Programming. In *Conference on Artificial Intelligence (AAAI)*.

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *User Interface Software & Technology (UIST)*. https://doi.org/10.1145/2807442.2807459

Çetin Meriçli, Steven D. Klee, Jack Paparian, and Manuela M. Veloso. 2014. An interactive approach for situated task specification through verbal instructions. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*.

George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995). https://doi.org/10.1145/219717.219748

Sara Mohammadinejad, Jyotirmoy V. Deshmukh, Aniruddh G. Puranic, Marcell Vazquez-Chanlatte, and Alexandre Donzé. 2020. Interpretable classification of time-series data using efficient enumerative techniques. In *Hybrid Systems: Computation and Control (HSCC)*. https://doi.org/10.1145/3365365.3382218

Daniel Neider and Ivan Gavran. 2018. Learning Linear Temporal Properties. In *FMCAD (Formal Methods in Computer Aided Design)*. https://doi.org/10.23919/FMCAD.2018.8603016

Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. In *International Conference on Machine Learning (ICML)*.

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3180155.3180189

Vittorio Perera and Manuela M. Veloso. 2015. Handling Complex Commands as Service Robot Task Requests. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Amir Pnueli. 1977. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. https://doi.org/10.1109/SFCS.1977.32

Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. In *International Conference on Learning Representations (ICLR)*.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J. Pappas, and Sanjit A. Seshia. 2016. Implan: Scalable Incremental Motion Planning for Multi-Robot Systems. In *International Conference on Cyber-Physical Systems (ICCPS)*. https://doi.org/10.1109/ICCPS.2016.7479105

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-21690-4_23

Jesse Thomason, Shiqi Zhang, Raymond J. Mooney, and Peter Stone. 2015. Learning to Interpret Natural Language Commands through Human-Robot Dialog. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Marcell Vazquez-Chanlatte, Susmit Jha, Ashish Tiwari, Mark K. Ho, and Sanjit A. Seshia. 2018. Learning Task Specifications from Demonstrations. In *Neural Information Processing Systems (NeurIPS)*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017a. Interactive Query Synthesis from Input-Output Examples. In *SIGMOD Conference*. https://doi.org/10.1145/3035918.3058738

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017b. Synthesizing highly expressive SQL queries from input-output examples. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3140587.3062365

Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. 2017c. Naturalizing a Programming Language via Interactive Learning. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. https://doi.org/10.18653/v1/P17-1086

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017). https://doi.org/10.1145/3133887