Auton Robot (2015) 38:89–105
DOI 10.1007/s10514-014-9418-8

Provably correct reactive control from natural language

Constantine Lignos · Vasumathi Raman · Cameron Finucane · Mitchell Marcus · Hadas Kress-Gazit

Received: 1 October 2013 / Accepted: 8 October 2014 / Published online: 19 November 2014

Abstract This paper presents an integrated system for generating, troubleshooting, and executing correct-by-construction controllers for autonomous robots using natural language input, allowing non-expert users to command robots to perform high-level tasks. This system unites the power of formal methods with the accessibility of natural language, providing controllers for implementable high-level task specifications, easy-to-understand feedback on those that cannot be achieved, and natural language explanation of the reason for the robot's actions during execution. The natural language system uses domain-general components that can easily be adapted to cover the vocabulary of new applications. Generation of a linear temporal logic specification from the user's natural language input uses a novel data structure that allows for subsequent mapping of logical propositions back to natural language, enabling natural language feedback about problems with the specification that are only identifiable in the logical form. We demonstrate the robustness of the natural language understanding system through a user study where participants interacted with a simulated robot in a search and rescue scenario. Automated analysis and user feedback on unimplementable specifications is demonstrated using an example involving a robot assistant in a hospital.

C. Lignos (✉) · M. Marcus
Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA 19104, USA
e-mail: constantine@lignos.org

V. Raman
Department of Computing and Mathematical Sciences, California Institute of Technology, 1200 E California Blvd, Pasadena, CA 91125, USA

C. Finucane · H. Kress-Gazit
Sibley School of Mechanical and Aerospace Engineering, Cornell University, 105 Upson Hall, Ithaca, NY 14853, USA

Keywords Natural language · Formal methods · High-level control · Synthesis

1 Introduction

Natural language control of robots can enable the integration of robots into more diverse settings such as homes and offices and reduce the cognitive load of controlling robots in demanding situations such as urban search and rescue. To achieve such broad utility for these systems, untrained or minimally-trained users should be able to express task specifications and receive a response from the system in natural language, explaining what actions the robot will undertake and any issues that may be present in the specification.

The challenge of programming robots to perform these tasks has until recently been the domain of experts, requiring hard-coded high-level implementations and ad-hoc use of low-level techniques such as path-planning during execution. Recent advances in the application of formal methods for robot control have enabled automated synthesis of correct-by-construction hybrid controllers for complex high-level tasks (e.g., Kloetzer and Belta 2008; Karaman 2009; Bhatia et al. 2010; Bobadilla et al. 2011; Kress-Gazit et al. 2009; Wongpiromsarn et al. 2010). However, most current approaches require the user to provide task specifications in logic or a similarly structured specification language. This forces users to formally reason about system requirements rather than state an intuitive description of the desired outcome. Furthermore, the outcome of verifying such specifications is traditionally simply a response of success or failure; detecting which portions of the specification are at fault is a non-trivial task (e.g., Raman and Kress-Gazit 2013a), as is explaining the problem to the user.

The system presented in this paper combines the power of formal methods with the accessibility of natural language, providing correct-by-construction controllers for specifications that can be implemented and easy-to-understand feedback for those that cannot. The system is open-source, can be extended to cover new scenarios, and allows for both specification and execution of natural language specifications. The system enables users to specify high-level behaviors via natural language, parsing commands using semantic analysis to create a linear temporal logic (LTL) specification. This parsing is deterministic and predictable, providing feedback to the user to help guide them if their input could not be completely parsed. The LTL specification is used to synthesize a hybrid controller when possible. If no implementation exists, the user is provided with an explanation and the portions of the specification that cause failure. The explanation is enabled by a data structure called a *generation tree*, which records the transformation of natural language into formal logic and is able to map backwards from problematic logical propositions to the explicit or implicit input from the user that caused the problem.

The system allows for execution of the controller either in simulation or using a physical robot. Natural language features are also available during execution; the generation tree allows the robot to explain what goal it is trying to achieve through its current action. The same language processing and semantic extraction components that enable the creation of logical propositions from natural language specifications enable feedback from the robot during execution.

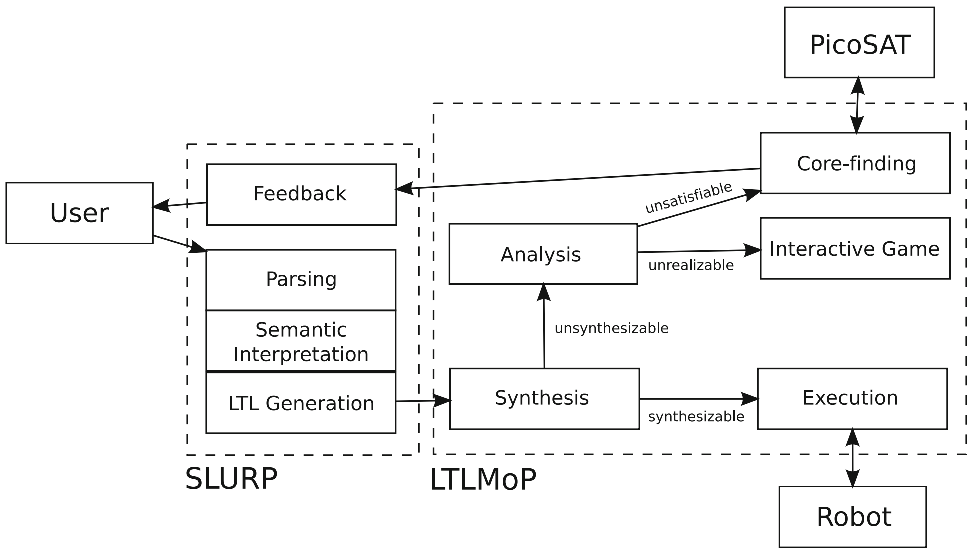
Figure 1 shows the system's main components and the connections between them. The Situated Language Understanding Robot Platform (*SLURP*) consists of parsing, semantic interpretation, LTL generation, and feedback components. This module is the natural language connection between the user and the logical representations used within the Linear Temporal Logic MissiOn Planning (*LTLMoP*) toolkit (Finucane et al. 2010). LTLMoP, which also interfaces with the SAT solver PicoSAT (Biere 2008), provides an environment for creating, analyzing, and executing specifications.

This organization of this paper follows the flow of user input into the system. Section 2 discusses previous work in this area. Section 3 describes the language understanding and LTL generation mechanisms and presents an evaluation of the performance of the system when processing commands from users. Sections 4 and 5 describe controller synthesis and specification feedback mechanisms and present an example of applying the system in a hospital setting. Discussion of the system and future work follows in Sect. 6.

2 Related work

There are many previous approaches that use natural language for controlling robots, differing in the type of information they seek to extract from natural language, the type of dialog desired, and the role of learning. To produce formal goal descriptions and action scripts for tasks like navigation and manipulation, Dzifcak et al. (2009) use a combinatorial categorial grammar (CCG) parser with a pre-specified mapping between words or phrases and the matching branching temporal logic and dynamic logic propositions. Another area of focus is language grounding scenarios where language must be mapped to objects or locations (e.g., Matuszek et al. 2012; Tellex et al. 2011), learning the relationship between the words used and the referents in the world.

Fig. 1 System overview



Recent work has focused on the automatic learning of the relationship between language and semantic structures, both in general semantic parsing (e.g., Berant and Liang 2014; Poon and Domingos 2009) and robotics-specific applications (e.g., Chen and Mooney 2011; Matuszek et al. 2010, 2012, 2013). While the learning strategies used may be generalized to broader applications, these systems are typically trained and evaluated in a single command domain (e.g., navigation, manipulation) and typically verify understanding at the per-utterance level or a single action sequence such as sequential navigation commands.

In contrast to learning-focused work, the system presented here focuses on the construction of a complete, formally-verified specification from natural language, but assumes the grounding between language and the robot's capabilities can be specified in advance. In doing so, we explore the broader relationships between natural language and logical form and the challenges involved in creating rich specifications from natural language that can include any number of action types. To be applied to the kinds of high-demand scenarios where natural language control may be of greatest benefit, it is essential that the design of the system be centered around "failing fast" by reporting any errors before execution.

This work aims to generate controllers for autonomous robots that achieve desired high-level behaviors, including reacting to external events and repeated patrol-type behaviors. Examples of such high-level tasks include search and rescue missions and the control of autonomous vehicles following traffic rules in a complex environment, such as in the DARPA Urban Challenge. With the usual approach of hard-coding the high-level aspects and using path-planning and other low-level techniques during execution, it is often not known a priori whether the proposed implementation actually captures the high-level requirements. This motivates the application of formal frameworks to guarantee that the implemented plans will produce the desired behavior.

A number of frameworks have recently been proposed, some of which use model checking (Clarke et al. 1999) to synthesize control laws (e.g., Kloetzer and Belta 2008; Bhatia et al. 2010) on a discrete abstraction of the underlying system. Other approaches, such as those proposed by Kress-Gazit et al. (2009) and Wongpiromsarn et al. (2010), apply efficient synthesis techniques to automatically generate provably-correct, closed-loop, low-level robot controllers that satisfy high-level reactive behaviors specified as LTL formulas. Specifications describe the robot's goals and assumptions regarding the environment it operates in, using a discrete abstraction. The hybrid robot controllers generated represent a rich set of infinite behaviors, and the closed loop system they form is guaranteed to satisfy the desired specification in any admissible environment, one that satisfies the modeled assumptions.

Previous work using LTL synthesis for robot control has also used highly structured or domain-specific languages to allow non-technical users to write robot specifications, even if they are unfamiliar with the underlying logic. For example, LTLMoP includes a parser that automatically translates sentences belonging to a defined grammar into LTL formulas (Kress-Gazit et al. 2008; Finucane et al. 2010); the grammar includes conditionals, goals, safety sentences and non-projective locative prepositions such as *between* and *near*. Structured English circumvents the ambiguity and computational challenges associated with natural language, while still providing a more intuitive medium of interaction than LTL. However, users still need to understand many details of the logical representation and the synthesis process to successfully write specifications in structured English. The work presented in this paper replaces LTLMoP's structured English input with a natural language interface to enable users to describe high-level tasks in more natural language.

Recent work has also tackled the problem of analyzing high-level specifications that are unsynthesizable. For example, Cizelj and Belta (2013) present a system based on human-supervised specification updates according to pre-specified rules. The focus in this paper, on the other hand, is on automated specification analysis. In contrast to that work, we present methods for reactive LTL specifications rather than probabilistic computation tree logic.

Feedback about the cause of unsynthesizability can be provided to the user in the form of a modified specification (Fainekos 2011; Kim et al. 2012), a highlighted fragment of the original specification (Raman and Kress-Gazit 2011), or by allowing the user to interact with an adversarial environment that prevents the robot from achieving the specified behavior (Raman and Kress-Gazit 2013a). This work follows the approaches introduced by Raman et al. (2013) and Raman and Kress-Gazit (2013b) to provide minimal explanations for unsynthesizable specifications, as described in Sect. 4. The system presented in this paper provides fine-grained natural language feedback that is not necessarily a subset of the original natural language specification. It also enhances the interactive visualization tool introduced in Raman and Kress-Gazit (2013a) with feedback on why particular robot actions may be disallowed in a given state, as described in Sect. 5.2.

This paper extends work presented by Raman et al. (2013) by analyzing the performance of the natural language understanding system through a user study and adding additional LTL generation features, support for feedback for all types of unsynthesizable specifications, and further discussion regarding the design of the integrated system. We provide a more complete approach to generating LTL from natural language input (Sect. 3), addressing the issues of non-compositionality when considering negation over natural language and providing results from a user study that demonstrates the system's robustness when used by inexperienced users. To allow feedback in all cases where the specification is unsynthesizable, we extend the natural language feedback to *unrealizable* specifications (Sect. 4) by incorporating the core-finding techniques recently introduced by Raman and Kress-Gazit (2013b), Raman and Kress-Gazit (2014) and mapping the logical causes of failure to natural language to provide feedback to the user.

3 Transforming natural language into logic with SLURP

To convert the user's commands into a formal specification, the system must identify the underlying linguistic structure of the commands and convert it into a logical representation, filling in appropriate implicit assumptions about the desired behavior. This section describes the process of this conversion and its implementation as The Situated Language Understanding Robot Platform (SLURP). SLURP enables the conversion of natural language specifications into LTL formulas, communication with the user regarding problems with specifications, and feedback to the user during execution. Sections 3.2–3.3 discuss the process of generating LTL formulas from natural language input. Section 3.4 describes a user study used to evaluate the performance of the system described.

3.1 Overview

In using the term *natural language*, we refer to language that a user of the system would produce without specific knowledge of what the system is capable of understanding. In other words, language that is not restricted to a known set of vocabulary items or grammatical structures specific to the system. Users are able to give commands without any knowledge of how the language understanding system works, as if they were giving simple, clear instructions to another person. While the user may be able to give commands to the system that it cannot understand, the system gives feedback based on what it understood and what it did not. For example, it may report that it does not understand how to carry out the verb used in a command, or may report that it does know how to perform that verb but has not received sufficient information, for example being told to move but not told where to do so.

The user's instructions are processed through a pipeline of natural language components similar to that used by Brooks et al. (2012) which identify the syntactic structure of the sentences, extract semantic information from them, and create logical formulas to be used in controller synthesis. While many previous natural language systems for robot control have relied on per-scenario grammars that combine semantic and syntactic information (e.g., Dzifcak et al. 2009), this work uses a combination of robust, general-purpose components for tagging and parsing the input. An advantage of this approach compared to per-scenario grammars is that the core language models need not be modified across scenarios; to adapt to new scenarios all that is required is that the LTL generation be extended to support additional types of commands. This reduces the role of the fragile process of grammar engineering and minimizes the cost of adapting the system to handle commands in new domains.

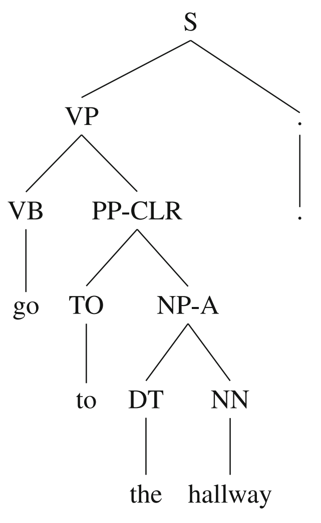
3.2 Identifying linguistic structure

Before a sentence may be converted into logical formulas, the linguistic structure of the sentence must be identified. Following traditional practices in natural language processing, this process is divided into modules: first, the syntactic structure of the input is extracted; second, the meaning of the sentence is recovered by identifying verbs and their arguments. These steps are explained in detail in Sects. 3.2.1 and 3.2.2.

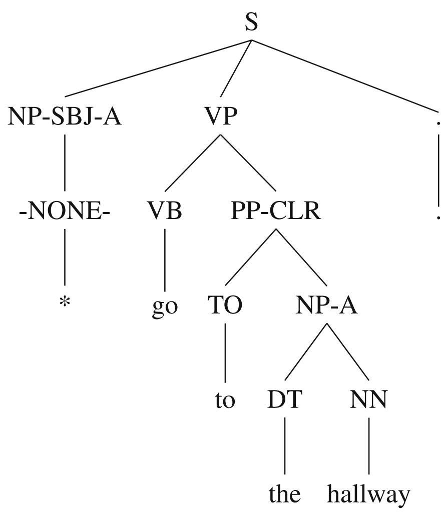
3.2.1 Parsing and null element restoration

Parsing is the process of assigning a hierarchical structure to a sentence. While simple natural language understanding can be performed with shallower processing techniques, parsing allows for recovery of the hierarchical structure of the sentence, allowing for proper handling of natural language phenomena such as negation (e.g., *Never go to the lounge*) and coordination (e.g., *Go to the lounge and kitchen*), which are crucial to understanding commands. SLURP uses a pipeline of domain-general natural language processing components. The input is tagged using the Stanford Log-linear Part-Of-Speech Tagger (Toutanova et al. 2003) and parsed using the Bikel parser (Bikel 2004); these parses are then post-processed using the null element (understood subject) restoration system of Gabbard et al. (2006). The models used by these systems require no in-domain training. An example output of these modules is given in Fig. 2a, b.

The use of null element restoration, a step typically ignored in NLP systems, allows for correct parsing of imperatives and questions, critical structures for natural language control and dialog systems. For example, in Fig. 2a the original parse contains no subject at all as there is no overt subject in the input sentence. Figure 2b shows that null element restoration has added an understood subject marked by $*$. This allows the structure of imperatives to be straightforwardly matched by the semantic interpretation module, which will look for verbs by identifying subtrees that are rooted by *S*, and contain a subject (*NP-SBJ*) and a verb phrase (*VP*). After null element restoration, an imperative has the same underlying structure as a statement with an explicit subject (e.g., *A patient is in r1*), allowing a general-purpose semantic interpretation system to process all input without using adhoc techniques to accommodate imperatives.



(a) Tagging and parsing



(b) Null element restoration

Agent: * (understood subject)
Verb: go
Preposition: to
Location: the hallway

(c) VerbNet frame matching

Initially, the hallway has not been visited:
$\neg s.mem\_visit\_hallway$
Define a persistent memory of going to the hallway:
$\Box(\bigcirc s.mem\_visit\_hallway \Leftrightarrow (s.mem\_visit\_hallway \vee \bigcirc s.hallway))$
Always eventually have a memory of visiting the hallway:
$\Box\Diamond(s.mem\_visit\_hallway)$

(d) LTL formula generation

Fig. 2 Conversion of the sentence *Go to the hallway* into LTL formulas through tagging, parsing, null element restoration, semantic interpretation, and LTL generation

3.2.2 Semantic interpretation

The semantic interpretation module uses the parse tree to extract verbs and their arguments. For example, in the sentence *Carry meals from the kitchen to all patient rooms*, the desired structure is a *carry* command with an object of *meals*, a source of *kitchen*, and a destination of *all patient rooms*. Objects identified may be further processed, for example *all* will be identified as a quantifier and handled as described in Sect. 3.3.2.

To identify verbs and their arguments in parse trees, SLURP uses VerbNet (Schuler 2005), a large database of verbs and the types of arguments they can take. The VerbNet database identifies verbs as members of senses: groups of verbs which in similar contexts have similar meanings. For example, the verbs *carry*, *lug*, and *haul* belong to the sense CARRY, because in many contexts they are equivalent in meaning. For each sense, VerbNet provides a set of frames, which indicates the possible arguments to the sense.

In the preceding example sentence, the verb *carry* is mapped to the sense CARRY. An example of a frame for this sense is (*Agent, Verb, Theme, Source, Destination*), thus the expected use of the verb is that there is someone performing it (*Agent*), someone or something it is being performed on (*Theme*), and path to perform it on (*Source* and *Destination*). Each role in the frame, subject to its associated syntactic constraints, is mapped to a part of the parse tree. VerbNet only provides information about the verb arguments; SLURP matches these argument types by mapping them to part-of-speech (e.g., *VB* for base verb) and phrasal (e.g., *NP* for noun phrase) tags and identifying each argument using its tag and syntactic position. For the above example, SLURP creates the following mapping:

Agent	the robot (understood subject)
Verb	carry
Theme	meals
Source	the kitchen
Destination	all patient rooms

Among the frames that match the parse tree, SLURP chooses the frame that expresses the most semantic roles. For example, the CARRY sense also contains a frame (*Agent, Verb, Theme, Destination*), which may be used in cases where the source is already understood, for example if the user previously stated *The meals are in the kitchen*. However, this frame will not be selected in the above example because the more specific frame that contains a source—and thus expresses more semantic roles—also matches. The chosen match is then used to fill in the appropriate fields in the command.

Matching of frames is not limited to entire sentences. In a sentence such as *If you see an intruder, activate your camera*, frames are matched for both the conditional clause and the main clause, allowing for the condition (*Agent*: the robot, *Verb*: see, *Theme*: an intruder) to be applied to the action (*Agent*: the robot, *Verb*: activate, *Theme*: your camera) when generating the logical representation.

3.3 Linear temporal logic generation

The information provided by VerbNet allows the identification of verbs and their arguments; these verbs can then be used to generate logical formulas defining robot tasks.

3.3.1 Linear temporal logic

The underlying logical formalism used in this work is linear temporal logic (LTL), a modal logic that includes temporal operators, allowing formulas to specify the truth values of atomic propositions over time. Let $AP = \mathcal{X} \cup \mathcal{Y}$, where $\mathcal{X}$ is the set of "input" propositions, those controlled by the environment, and $\mathcal{Y}$ is the set of "output" propositions, those controlled by the robot. LTL formulas are constructed from atomic propositions $\pi \in AP$ according to the following recursive grammar:

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi\, \mathcal{U}\, \varphi,$$

where $\neg$ is negation, $\vee$ is disjunction, $\bigcirc$ is "next", and $\mathcal{U}$ is a strong "until." Conjunction ($\wedge$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), "eventually" ($\Diamond$) and "always" ($\Box$) are derived from these operators. Informally, the formula $\bigcirc \varphi$ expresses that φ is true in the next time step. Similarly, a sequence of states satisfies $\Box \varphi$ if φ is true in every position of the sequence, and $\Diamond \varphi$ if φ is true at some position of the sequence. Therefore, the formula $\Box \Diamond \varphi$ is satisfied if φ is true infinitely often. For a formal definition of the LTL semantics, see Clarke et al. (1999).

Task specifications in this work are expressed as LTL formulas from the fragment known as *generalized reactivity of rank 1* (GR(1)), and have the form $\varphi = \varphi_e \Rightarrow \varphi_s$ with $\varphi_p = \varphi_p^i \wedge \varphi_p^t \wedge \varphi_p^g$, where φ_p^i, φ_p^t and φ_p^g for $p \in \{e, s\}$ represent the initial conditions, safeties and goals, respectively, for the environment (e) and the robot (s). The restriction to GR(1) is for computational reasons, as described in Kress-Gazit et al. (2009).

3.3.2 Types of commands

There are two primary types of properties allowed: *safety* properties, which guarantee that "something bad never happens," and *liveness* conditions, which state that "something good (eventually) happens." These correspond naturally to LTL formulas with operators $\Box$ and $\Diamond$. While the domain of actions expressible in natural language is effectively infinite, the set of actions that a robot can perform in practice is limited. Examples of semantic behaviors currently implemented include:

1. Actions that need to be completed once, for example going to rooms (*Go to the hallway.*)
2. Actions that need to be continuously performed, for example patrolling multiple areas (*Patrol the hallway and office.*)
3. Completing a long-running action that can be interrupted, for example searching a room and reacting to any items found (*Search the hallway.*)
4. Following (*Follow me.*)
5. Enabling/disabling actuators (*Activate your camera.*)
6. Carrying items (*Carry meals from the kitchen to all patient rooms.*)

Each command is mapped to a set of senses in VerbNet so that a varied set of individual verbs may be used to signify each command. As a result, SLURP is only limited in its vocabulary coverage by the contents of VerbNet—which is easily expanded to support additional verbs if needed—and by what actions can be transformed into LTL. While the number of syntactic structures identifiable by the system is unbounded, the set of frames that SLURP can recognize and transform into logical form is constrained by the mapping of frames to robot actions.

Each command may be freely combined with conditional structures (*If you hear an alarm...*), negation (*Don't go to the lounge*), coordination (*Go to the hallway and lounge*), and quantification (*Go to all patient rooms*). The use of quantification requires that, in constructing the scenario, information about quantifiable sets is specified; for example, the command "go to all patient rooms" can be unrolled to apply to all rooms that have been tagged with the keyword *patient*.

3.3.3 Generation

For each supported command, LTL is generated by macros which create the appropriate assumptions, restrictions, and goals. In the example given in Fig. 2, the resulting LTL formulas define a memory of having visited the hallway, and the goal of setting that memory.[1]

Formulas are generated by mapping each command to combinations of macros. These macros include:

1. Goals: $goal(x)$ generates $\Box \Diamond (x)$
2. Persistent memories: $memory(x)$ generates $\Box(\bigcirc s.mem\_x \Leftrightarrow (s.mem\_x \vee \bigcirc s.x)))$
3. Complete at least once (ALO): $alo(x)$ generates $(goal(s.mem\_x) \wedge memory(x))$

Among the simplest commands to generate are those that are directly mapped to goals; i.e., ones that are performed

[1] This arguably unintuitive translation is due to specifications in LTLMoP being restricted to the GR(1) fragment of LTL.

infinitely often. For example, patrolling a room maps to $goal(room)$; if multiple rooms are to be patrolled, execution will satisfy each goal in turn, moving the robot from room to room indefinitely.

Commands for which there is a distinct notion of completion—in linguistic terminology, commands which contain a verb of perfective aspect—typically generate persistent memories. For example, *Go to the hallway* is interpreted as visit—go to *at least once*—the hallway: $alo(hallway)$. In this case, the robot's goal is to have a "memory" of having been in the hallway; this memory proposition is set by entering the hallway, after which the memory persists indefinitely and the goal is trivially satisfied from then on.

A challenge in creating a correct mapping is that the negation of a command does not necessarily imply its logical negation. For example, *Don't go to the hallway* is most concisely expressed as the safety $\square\neg s.hallway$ (literally, *Always, do not be in the hallway*), as opposed to specifying that the robot should infinitely often achieve a goal of not having a memory of being in the hallway. In this case, the negation of a goal yields a safety. In general, negation of a sentence in natural language does not always transparently propagate to a negation of the logical statement that the positive form of the sentence would have generated.

However, for commands that create safeties, negation is much simpler. For example, *If you see an intruder, activate your camera* becomes $\square(\bigcirc e.intruder \Rightarrow \bigcirc s.camera)$, and the negated form *If you see an intruder, do not activate your camera* becomes $\square(\bigcirc e.intruder \Rightarrow \bigcirc\neg s.camera)$. Negative commands are always expressed as safeties, while the positive version of the same command may not be, as in the previous example of *Go to the hallway*. More complex examples of generation involving combinations of these macros are given in Sect. 5.

Support for new commands can be added by first verifying the presence of the verb and the intended argument structure in VerbNet, adding the verb and information about the arguments it takes if needed by editing an XML database. For example, a bomb-defusing robot will need to understand the verb *defuse*, which is not contained in VerbNet. Many new commands to be added can easily be expressed using the macros for *goal* or *alo* or mapping directly to an actuator on the robot. These commands can trivially be supported by the system by marking that sense as a simple action in the LTL generation subsystem and giving the macro it is mapped to. For example, for the user study reported in Sect. 3.4, we added a *defuse* sense to VerbNet and mapped it to a simulated actuator of the same name.

Complex high-level commands like the carry example given in Sect. 5 require the user to explicitly specify the LTL to be generated by the command. For example, to represent the notion of carrying an object, safeties must be generated to reflect that the robot should pick up in the source location, drop off in the destination location, and can only pick up when it is not holding something and only drop when it is.

3.3.4 Generation tree

A novel aspect of the LTL generation process is that the transformations undertaken are automatically recorded in a *generation tree* to allow for a more interpretable analysis of the specification generated. As shown in Fig. 3, the generation tree allows for a hierarchical explanation of how LTL formulas are generated from natural language. There is a tree corresponding to each natural language statement, rooted at the natural language statement and with LTL formulas as leaves. The intermediate nodes are automatically created by the LTL generation process to explain how the statement was subdivided and why each LTL formula was generated.

In addition to allowing the user to inspect the generated LTL, the generation tree enables mapping between LTL formulas and natural language for specification analysis. As is shown in the following sections, this allows for natural language explanations of problems detected in the specification. During execution of the generated controller (either in simulation or with a real robot), it also allows the system to answer the question *What are you doing?* by responding with language from the generation tree. For example, in Fig. 3, if the current goal being pursued during execution is $\square\diamondsuit s.mem\_visit\_lounge$, the system responds: *I'm currently trying to "visit lounge."* In cases where the original instruction involves quantification, identification of the subgoal is particularly useful. If the user enters *Go to all patient rooms,* the generation tree will contain a sub-tree for each patient room, allowing for clear identification of which room is relevant to any problems with the specification.

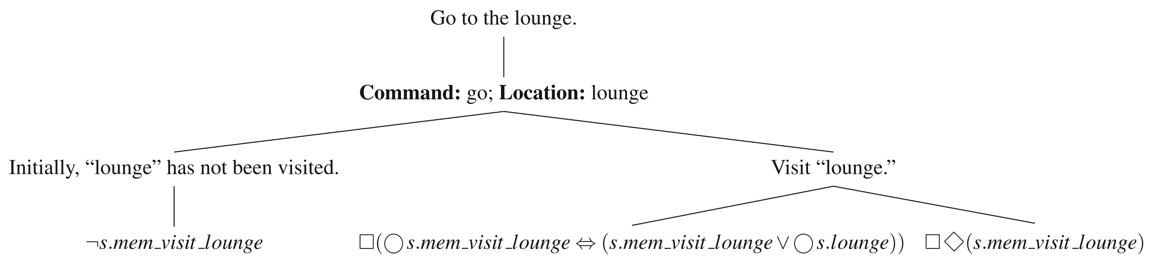
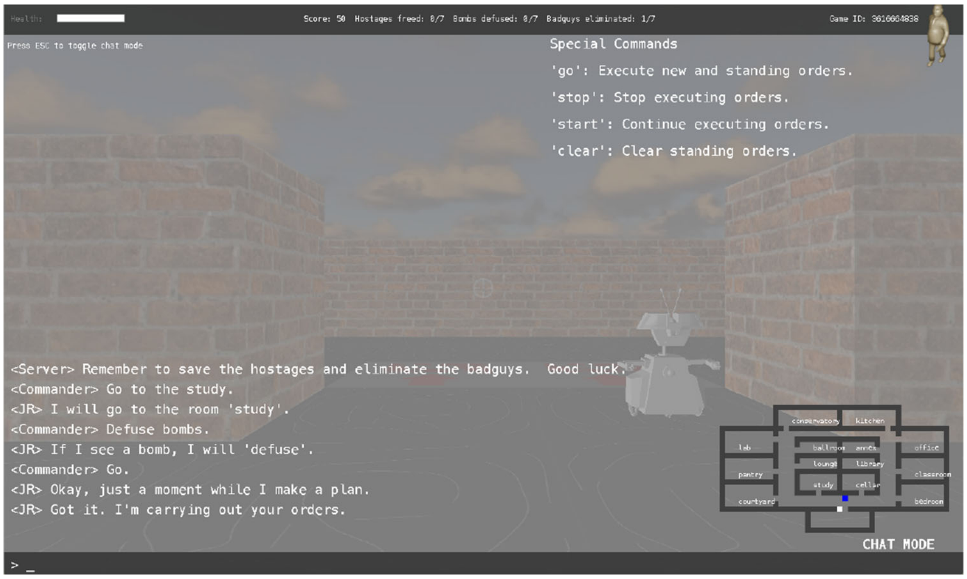

Go to the lounge.
Command: go; **Location:** lounge
Initially, "lounge" has not been visited.
Visit "lounge."
$\neg s.mem\_visit\_lounge$
$\square(\bigcirc s.mem\_visit\_lounge \Leftrightarrow (s.mem\_visit\_lounge \vee \bigcirc s.lounge))$
$\square\diamondsuit(s.mem\_visit\_lounge)$

Fig. 3 Generation tree for *Go to the lounge*



```
Special Commands
'go': Execute new and standing orders.
'stop': Stop executing orders.
'start': Continue executing orders.
'clear': Clear standing orders.
```

```
<Server> Remember to save the hostages and eliminate the badguys.  Good luck.
<Commander> Go to the study.
<JR> I will go to the room 'study'.
<Commander> Defuse bombs.
<JR> If I see a bomb, I will 'defuse'.
<Commander> Go.
<JR> Okay, just a moment while I make a plan.
<JR> Got it. I'm carrying out your orders.
> _
```

Fig. 4 Screenshot of simulation environment

3.4 Evaluation

3.4.1 Design

To evaluate the performance of this system when used by inexperienced users, we embedded SLURP as a robot agent in a first-person-perspective 3D video game, simulating a search and rescue scenario. The participant played the role of an operator instructing a robot through natural language commands. A screenshot of the game during an interaction with the robot is given in Fig. 4.

To succeed in the scenario, the operator was required to work with the robot to search 14 rooms on a floor, using the robot to defuse any bombs discovered while the operator rescued hostages. To increase the difficulty of the task and force greater reliance on the robot for searching, the operator needed to neutralize hostage takers who are trying to escape from the floor while it is searched. The scenario was considered a failure if the operator ever entered a room with an active (not yet defused) bomb or if too many of the hostage takers escaped. To succeed, the participant needed to command the robot to perform two tasks: navigate all rooms, and defuse all bombs found.

The user study was designed to elicit natural language commands from users without giving any explicit suggestions regarding what they should say to the robot and what commands it understood. After providing informed consent, participants were instructed that the simulated robot ("Junior") was capable of understanding natural commands and advised to communicate with it naturally ("Talk to Junior like you might talk to someone who needs instructions from you"), giving direct commands but not doing anything unnatural such as removing prepositions and articles or using "Tarzan-speak." The experimenter was permitted to answer questions about the instructions, game controls, and interface during the training scenarios, but not during the testing scenario. The experimenter did not give any suggestions regarding what the user should say to the robot or intervene in the experiment other than restarting the experiment in case of software failure.

Users participated in four training scenarios of increasing difficulty before attempting the full scenario described above. The purpose of the training scenarios was to introduce them to the game dynamics as well as give them simple tasks to perform with natural language before trying to complete more complex ones. The training scenarios used smaller maps and gradually introduced the actions that would need to be performed in the main scenario: neutralizing hostage takers, and commanding the robot to move between rooms and defuse bombs. The layout used in the final scenario contained 14 rooms connected by hallways in a ring configuration, a layout not used in any of the training exercises that made neutralizing the escaping hostage takers significantly more difficult. The locations of bombs, hostages, and hostage takers were randomly generated for each user and could be discovered by navigating to the room they were located in.

3.4.2 Results

All 14 participants successfully completed the training and testing scenarios. They were able to successfully command the robot navigate to and defuse all bombs present in the map.

We analyzed transcripts of interactions between users and the simulated robot across the four training scenarios and the final testing scenario. The transcripts consisted of 628 commands given to the simulated robot. 69 commands (11.0 %) were excluded from analysis for the following reasons: typographical errors, humorous commands unrelated to the

Table 1 User study performance

Overall performance

Result	Count	% of commands
Understood	526	94.1
Error	33	5.9

scenario (e.g., ordering the robot to dance), non-commands (e.g., telling the robot "good job"), commands the robot cannot perform in the scenario (e.g., instructing it to move to locations not on the map), unnecessary repetition of previously not-understood commands, or if software limitations of the simulation, not the language understanding system, caused the command to not be processed. The remaining 559 commands were automatically labeled for whether they were successfully understood by the system based on its response.

526 commands (94.1 %) were understood and resulted in the successful synthesis and execution of an automaton (Table 1). The 33 commands (5.9 %) that were not understood were manually annotated and investigated to determine the cause of the failure (Table 2). The largest single cause of errors was the tagger failing to identify imperatives as verbs because they are rarely present in the training data for standard natural language processing components. For example, both *Rescue the hostage* and *Free the hostage* were not recognized as imperatives due to tagging errors.[2]

The verbs the system failed to recognize were *come*, *destroy*, *find*, *get*, and *walk*. The failure to recognize *come* (*come here*) and *get* (*get up*) was caused by VerbNet not including the specific imperative uses of these verbs. The system was able to semantically parse sentences containing *destroy* and *walk*, but the VerbNet senses for these verbs had not been mapped to the *defuse* and *go* actions that users expected. This can be addressed by simply adding these mappings. In these cases, the user was informed that the system recognized the verb in their command but did not know what to do with it (*Sorry, but I don't know how to walk*), and the user discovered an alternate way to make their request (*go*, *move*), that was understood.

All but one instance of the errors due to use of shorthand of saying only *defuse* to instruct the system to defuse bombs. Successful commands for defusing bombs produced by users included *Defuse the bomb*, *Defuse bomb*, and compound commands such as *Defuse the bomb and go to the lab*. Two of the errors classified as "Other" were caused by the inability of the semantic parsing system to resolve pronouns within a command: *If there is a bomb, defuse it*.

[2] Even though it was the role of the operator, not the robot, to rescue hostages, we label these examples as tagging errors because a command was given to the system and it was not properly understood. The desired response in this situation is to understand the requested action but report that the robot cannot perform it.

3.4.3 Discussion

The evaluation presented here demonstrates that inexperienced users can successfully give commands understood by SLURP without any specific knowledge of what the system is capable of understanding or the underlying lexical database or parsing system. While users did produce commands the system did not understand, they were also able to identify alternative forms that worked and complete the scenarios. This was aided by the rich feedback the system provides. For example, when a user stated *If there is a bomb, defuse it*, the system's response was *Sorry, I don't understand what you mean by 'it'*. This response allows the user to identify the issue with their command and revise it. Systems that rely on a formal grammar for parsing (e.g., Dzifcak et al. 2009) report a parsing failure in this instance, not giving the user any information regarding what the issue is.

The majority of errors encountered were due to a common problem in natural language processing: differences in the type of content in the data that the tagger and parser were trained to perform well on and the actual data used in testing. The tagger and parser used are primarily trained on newswire, which contains a very small number of imperatives. This is in contrast to the data set evaluated here, where every sentence is an imperative. The data collected in this user study can be used to allow training on more relevant in-domain data, allowing for a reduction in the number of tagger and parser errors.

With the usability of the natural language understanding system validated, in the following sections we return to the process of generating automata from LTL specifications and providing feedback based on the generation tree.

4 Identifying minimal unsynthesizable cores in LTL specifications

This section discusses the construction of provably-correct controllers from LTL specifications using the Linear Temporal Logic MissiOn Planning (LTLMoP) toolkit, and the analysis of specifications for which no such controllers exist.

4.1 Controller synthesis and execution

Given an LTL formula representing a task specification and a description of the workspace topology, the efficient synthesis algorithm introduced by Piterman et al. (2006) is used to construct an implementing automaton (if one exists). In combination with lower-level continuous controllers, this automaton is then used to form a hybrid controller that can be deployed on physical robots or in simulation. To obtain this hybrid controller, a transition between two discrete states is achieved by the activation of one or more low-level

Table 2 User study error analysis

Causes of errors			
Error type	Count	% of commands	Example
Tagging	10	1.8	*Rescue* tagged as a noun in *Rescue the hostages*
Syntactic parsing	1	0.2	Null element not restored in *Go to north rooms, then go to east rooms*
Semantic parsing	2	0.4	*Around* not recognized as an argument in *Turn around*
Verb not understood	9	1.6	VerbNet sense for *walk* not mapped to movement action
Use of shorthand	7	1.3	One-word commands such as *defuse* and *bedroom*
Other	4	0.7	*It* not understood in *If you see a bomb, defuse it*

continuous controllers (Kress-Gazit et al. 2009; Finucane et al. 2010).

4.2 Specification analysis

If an implementing automaton exists for a given specification, it is called *synthesizable*. Otherwise, it is *unsynthesizable*, and the algorithm presented in (Raman and Kress-Gazit 2013a) is used to automatically analyze the LTL formula and identify causes of failure. The analysis also presents an *interactive game* for exploring possible causes of unsynthesizability, in which the user attempts to fulfill the robot's task specification against an adversarial environment, provided by the tool. However, the granularity of the feedback provided by this algorithm is relatively coarse. For example, it will identify a contradiction within the system safety conditions, but cannot pinpoint the exact safeties that are contradicting.

Recent work by Raman and Kress-Gazit (2014) improved upon this analysis to provide minimal explanations of failure. The system presented in this paper applies these techniques to produce fine-grained natural language feedback on the cause of failure, by tracing the problem back to a minimal explanation. The highlights of the relevant approaches are reviewed here to provide context for their use within the presented system; further details and formal algorithms can be found in the aforementioned papers.

4.3 Types of unsynthesizability

Unsynthesizable specifications are either *unsatisfiable*, in which case the robot cannot succeed no matter what happens in the environment (e.g., if the task requires patrolling a disconnected workspace), or *unrealizable*, in which case there exists at least one environment that can prevent the desired behavior (e.g., if in the above task, the environment can disconnect an otherwise connected workspace, such as by closing a door). More examples illustrating the differences between the two cases can be found in Raman and Kress-Gazit (2013a).

In both cases, failure can occur in one of two ways: either the robot ends up in a state from which it has no valid moves (termed *deadlock*), or the robot is always able to make a move but one of its goals is unreachable without violating the specified safety requirements (termed *livelock*). In the context of unsatisfiability, an example of deadlock is when the system safety conditions contain a contradiction within themselves. Similarly, unrealizable deadlock occurs when the environment has at least one strategy for forcing the system into a deadlocked state.

Previous work produced explanations of unsynthesizability in terms of combinations of the specification components (i.e., initial, safety and liveness conditions) (Raman and Kress-Gazit 2011, 2013a). However the true conflict often lies in small subformulas of these components. Consider Specification 1 . It is clear that the safety requirement in (1) conflicts with the goal in (2), since in order to visit the kitchen one must in fact go there. However, the safety requirement in (3) is irrelevant, and should be excluded from any explanation of why this specification is unsatisfiable.

Specification 1 An example unsatisfiable specification

1. *Don't go to the kitchen* (part of φ_s^t)
2. *Visit the kitchen* (part of φ_s^g)
3. *Always activate your camera* (part of φ_s^t)

Note that this is a case of livelock: the robot can follow its safety conditions indefinitely by staying out of the kitchen, but is prevented from ever reaching its goal of visiting the kitchen.

The above example motivates the identification of small, minimal, "core" explanations of the unsynthesizability. Raman and Kress-Gazit (2014) draw inspiration from the Boolean satisfiability (SAT) literature to define an unsynthesizable core of a GR(1) LTL formula as follows. Let $\varphi_1 \preceq \varphi_2 (\varphi_1 \prec \varphi_2)$ denote that φ_1 is a subformula (strict subformula, respectively) of φ_2.

Definition 1 Given a specification $\varphi = \varphi_e \Rightarrow \varphi_s$, a *minimal unsynthesizable core* is a subformula $\varphi_s^* \preceq \varphi_s$ such that $\varphi_e \Rightarrow \varphi_s^*$ is unsynthesizable, and for all $\varphi_s' \prec \varphi_s^*$, $\varphi_e \Rightarrow \varphi_s'$ is synthesizable.

4.4 Detecting unsynthesizable cores

4.4.1 Unsatisfiable cores via SAT solving

Raman and Kress-Gazit (2014) analyze unsatisfiable components of the robot specification by leveraging tools for finding minimal unsatisfiable subformulas of propositional SAT instances. Informally, the method creates a propositional SAT instance corresponding to truth assignments to all variables in $\mathcal{X} \cup \mathcal{Y}$ over the first d time steps of an execution satisfying $\varphi_s^i \wedge \varphi_s^t$, incrementing d to encode increasingly longer sequences of truth assignments. This resulting SAT instance at each step is tested for satisfiability using an off-the-shelf SAT solver. If an unroll depth d is reached such that the resulting SAT instance is unsatisfiable, there is no valid sequence of actions that follows the safety to d time steps starting from the initial condition, thus signaling a deadlock situation. In the case of livelock, a fixed depth d is used, and an additional clause added to represent the goal being required to hold at the last time step.

If the resulting SAT instance is unsatisfiable, the SAT solver yields a minimal unsatisfiable subformula, which is then mapped back to the originating portions of the safety and initial formulas. The guilty portions of the LTL specifications are in turn mapped back to the originating natural language instructions, as described in Sect. 5.

4.4.2 Unrealizable cores via SAT solving

If the specification is unrealizable rather than unsatisfiable, the above techniques do not apply directly to identify a core. Note that in the case of unsatisfiability, the SAT instances created at each unroll depth correspond to truth assignments to *all* atomic propositions in AP, including those controlled by the robot (i.e. $\mathcal{Y}$) as well as those controlled by the environment (i.e. $\mathcal{X}$). Doing so allows the environment variables to be set arbitrarily when searching for a truth assignment that fulfills the robot specification. Since an unsatisfiable specification cannot be fulfilled in *any* environment, allowing the SAT solver to "set" the environment when searching for a solution does not affect the outcome of the satisfiability test, or its implications for the original LTL formula.

On the other hand, if the specification is satisfiable but unrealizable, there exist some sequences of truth assignments to the input variables that allow the system requirements to be met. Therefore, to produce an unsatisfiable Boolean formula, all sequences of truth assignments to the input variables that satisfy the environment assumptions must be considered for each unroll depth. In the worst case, the number of depth-d Boolean formulas generated before an unsatisfiable formula is obtained grows exponentially in d.

Considering all possible environment input sequences is not feasible; fortunately, the environment counterstrategy produced by the specification analysis in Raman and Kress-Gazit (2013a) in the case of unrealizability provides us with inputs that will cause the robot to fail to achieve its specified task. For deadlock, the blocking states in the environment counterstrategy provide inputs to The reader is referred to Raman and Kress-Gazit (2013b, 2014) for the details of how the environment counterstrategy can be used in several cases to constrain the environment variables and determine an unrealizable core using a SAT-based approach.

4.4.3 Unsynthesizable cores via iterated realizability tests

There are some cases in which the above SAT-based analysis does not apply, as described in Raman and Kress-Gazit (2014). In addition, in the case of livelock, determining the shortest unroll depth required to produce a meaningful core is challenging. In these cases alternative, more computationally expensive techniques can be used to find a minimal core. The iterated realizability testing algorithm presented by Raman and Kress-Gazit (2014) provably computes an unrealizable core for specifications of the form considered in this work. Informally, the algorithm iterates through the conjuncts in the robot safety formula, removing them one at a time and checking realizability of the identified goal (under the original environment assumptions). If removing a conjunct makes the resulting formula synthesizable, that conjunct is retained as part of the minimal unsynthesizable core; otherwise it is left out and the procedure repeats with the next conjunct. When all conjuncts in the robot safety have been tested in this manner, the remaining specification is an unsynthesizable core. The computational tradeoffs between the approaches in Sects. 4.4.1, 4.4.2 and 4.4.3 are described in more detail by Raman and Kress-Gazit (2014).

The specification analysis reviewed in this section returns, in the case of deadlock, a minimal subset of the system safety requirements that causes failure; for livelock, a single goal is returned in addition to a minimal subset of safeties. Given this subset of formulas that cause unsynthesizability, it remains to map this set back onto the original specification. For example, in the case of the structured English specifications supported by the LTLMoP toolkit (Finucane et al. 2010), this is done by highlighting the sentences that produced the corresponding LTL (Raman and Kress-Gazit 2011). The generation of natural language feedback from the identified portions of the LTL formula in SLURP is discussed in Sect. 5.

5 Providing users with specification feedback

This section describes two modes of communicating the cause of unsynthesizability deployed in the presented system and gives examples of how these issues are communicated in an example scenario involving a robot acting as an assistant in a small hospital.

5.1 Explaining unsatisfiable tasks

Giving users detailed feedback regarding *why* a task is unachievable is essential to helping them correct it. While better than simply reporting failure, returning the user a set of LTL formulas responsible for unsatisfiability is not enough to help them correct the natural language specification that generated it. To provide actionable feedback to the user, SLURP uses a combination of the user's own natural language along with structured language created during the LTL formula generation process to explain problems with the specified task.

Consider the example given above where the combination of the statements "Don't go to the kitchen" and "Visit the kitchen" results in an unsatisfiable specification. The minimal unsatisfiable core of the specification is as follows:

$$\Box(\neg s.kitchen)$$
$$\Box(\bigcirc s.mem\_visit\_kitchen \Leftrightarrow (s.mem\_visit\_kitchen \vee \bigcirc s.kitchen))$$
$$\Box\Diamond(s.mem\_visit\_kitchen)$$

To explain the conflict, the system lists the goal that cannot be satisfied and natural language corresponding to the safety formulas in the minimal core. The response is:

> The problematic goal comes from the statement 'Go to the kitchen.' The system cannot achieve the sub-goal "Visit 'kitchen'."
> The statements that cause the problem are:
>
> – "Don't go to the kitchen." because of item(s): "Do not go to 'kitchen'."
> – 'Go to the kitchen.' because of item(s): "Visit 'kitchen'."

Additional examples of feedback are given in Sect. 5.4.1.

5.2 Interactive exploration of unrealizable tasks

Succinctly summarizing the cause of an unrealizable specification is challenging, sometimes even for a human, so our system uses an interactive game to demonstrate environment behavior that will cause the robot to fail. The game lets the user attempt to play as a robot against an adversarial environment, and in the process gain insight into the nature of the problem.

At each discrete time step, the user is presented with the current goal to pursue and the current state of the environment. The user is then able to change the location of the robot and the states of its actuators in response. By using the core-finding analysis presented in this work, a specific explanation is now given about what part of the original specification is in conflict with any invalid moves. This is done by finding the unsatisfiable core of a single-step satisfiability problem involving the user's current state, the desired next state, and all of the robot's specified safety conditions. An example of this interactive game in use is given in Sect. 5.4.2, and a screen capture of the game is shown in Fig. 7.

5.3 Explanation of current behavior

When executing a complex specification, it may not always be apparent to the user why the robot is performing a particular action. The structure of the generation tree allows for explaining the robot's current goal the same way that it supports the identification of a goal that causes a problem in a specification. During synthesis, each state in the control automaton is marked with the index of the goal that is currently being pursued by the strategy (Piterman et al. 2006). The generation tree allows for reverse-mapping of each of these goals to the natural language input that generated it.

Specification 2 Example for behavior explanation

1. Go to the kitchen.
2. Start in the hallway.
3. Do not go to the dining room.

For example, consider Specification 2. Assume that the dining room is the most direct path to the kitchen, but there is another path from the hallway to the kitchen by passing through the study, a more circuitous route. While the cause of taking a longer route is clear in this small specification, in a more complex specification being able to ask the robot why it is doing the current action can help maintain the user's trust and understanding of the robot's plan. When asked what it is doing, in this example the system responds: "I'm currently trying to 'visit kitchen'." The language the robot reflects represents an intermediate stage of the generation tree generated from the user's input; it is not simply echoing the user's input but rather making its own understanding of the commands apparent.

5.4 Hospital example

The remainder of this section presents three examples that demonstrate various features of this framework. All of the scenarios concern a robot acting as an assistant in a small

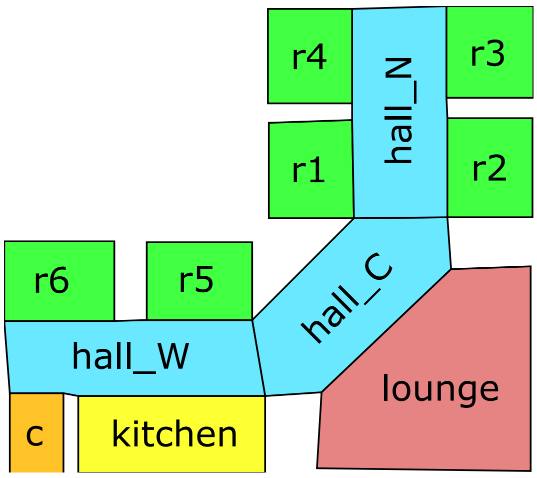


Fig. 5 Map of hospital workspace ("c" is the closet)

hospital (a map of the workspace is shown in Fig. 5). The robot is able to detect the location of the user, record video with its camera, and pick up and deliver objects.

5.4.1 Unsatisfiability

Specification 3 Example of unsatisfiability (deadlock)

1. Don't activate your camera in any restricted area.
2. Avoid the lounge.
3. Start in hall_c.
4. Always activate your camera.

Unsatisfiable *deadlock* can arise when the robot safety constraints are in direct conflict with one another. In Specification 3, the robot is given constraints to respect privacy in Lines 1 and 2 ("restricted areas" are defined as all rooms other than the lounge, closet, and kitchen), but is also asked to do something in direct contradiction with these constraints in Lines 3 and 4.

Even though the quantifier in Line 1 generates a large number of safety restrictions (one for each "restricted area"), the core-finding component correctly narrows down the problem and produces the following output:

The statements that cause the problem are:

- "Always activate your camera." because of item(s): "Always activate 'camera'."
- "Avoid the lounge." because of item(s): "Do not go to 'lounge'.", "The robot does not begin in 'lounge'."
- "Don't activate your camera in any restricted area." because of item(s): "Never activate 'camera' in 'hall_c'.", "Never activate 'camera' in 'hall_n'.", "Never activate 'camera' in 'hall_w'."
- "Start in hall_c." because of item(s): "The robot begins in 'hall_c'."

The obvious conflict identified by the system is that the robot must activate its camera when starting execution but it cannot immediately reach a non-restricted area from hall_c. *Avoid the lounge* is included in the core because otherwise the robot could simultaneously activate the camera and move from hall_c into the lounge, which is not a restricted area. Similarly, the inclusion of the starting position of hall_c is necessary because if the robot were to start in the closet, this specification would in fact be achievable by just staying in the closet and turning on the camera.

Specification 4 Example of unsatisfiability (livelock)

1. Start in the closet.
2. Carry meals from the kitchen to all patient rooms.
3. Don't go into any public rooms.

Unsatisfiable *livelock* is exhibited by the meal delivery mission shown in Specification 4, in which the robot is tasked with delivering meals to patients (in $r1$ to $r6$) while avoiding the "public rooms" (defined as $hall\_c$ and $lounge$). Because $hall\_c$ is considered a public room, a semantic subset of the safety requirement in Line 3 prevents the robot from being able to deliver meals to all of the patients as requested.

In addition to sentential feedback such as that shown for the previous example, the offending specification fragments are highlighted for the user in the context of the semantic LTL generation tree (see Fig. 6). Note that analysis always addresses only a single goal at a time, in this case choosing to highlight the reason that delivery to $r1$ is impossible.

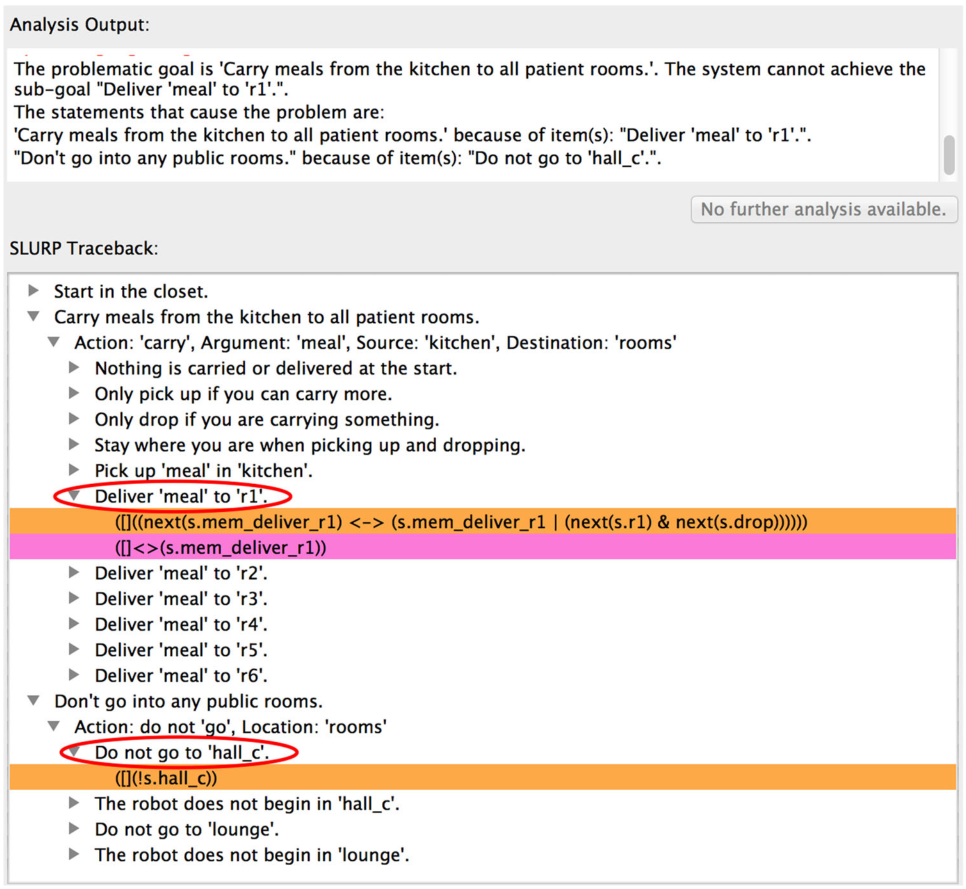
5.4.2 Unrealizability

As introduced in Sect. 5.2, unrealizable specifications can be analyzed using an interactive visualization tool (see Fig. 7). For example, in the case of Specification 5, we discover that the robot cannot achieve its goal of following the user (Line 1) if the user enters the kitchen (which the robot has been banned from entering in Line 2).

This conflict is presented to the user as follows: the environment sets its state to represent the target's being in the kitchen, and then, when the user attempts to enter the kitchen, the tool explains that this move is in conflict with Line 2.

Specification 5 Example of unrealizability

1. Follow me.
2. Avoid the kitchen.



Analysis Output:

The problematic goal is 'Carry meals from the kitchen to all patient rooms.'. The system cannot achieve the sub-goal "Deliver 'meal' to 'r1'.".
The statements that cause the problem are:
'Carry meals from the kitchen to all patient rooms.' because of item(s): "Deliver 'meal' to 'r1'.".
"Don't go into any public rooms." because of item(s): "Do not go to 'hall_c'.".

No further analysis available.

SLURP Traceback:

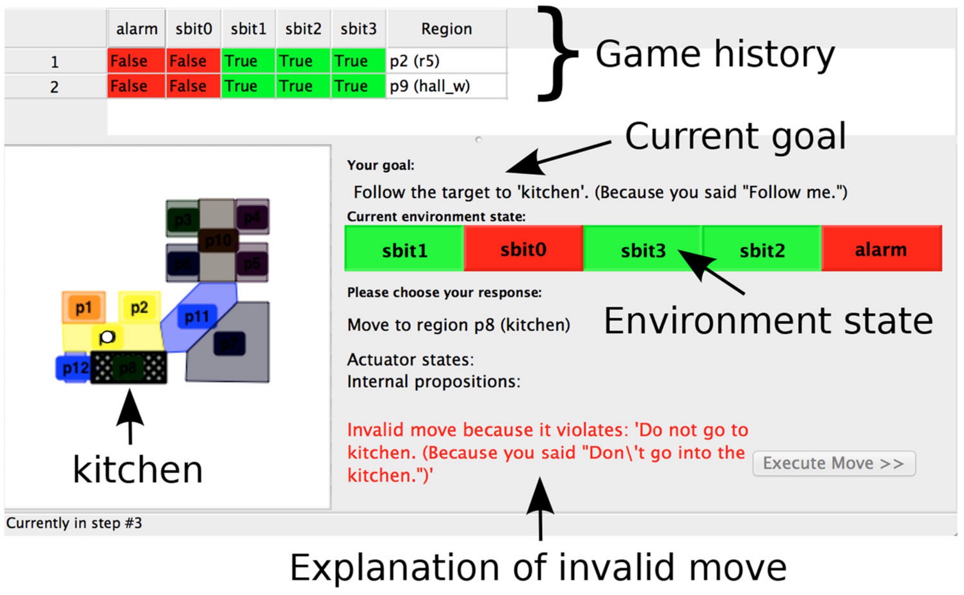
```
▶ Start in the closet.
▼ Carry meals from the kitchen to all patient rooms.
  ▼ Action: 'carry', Argument: 'meal', Source: 'kitchen', Destination: 'rooms'
    ▶ Nothing is carried or delivered at the start.
    ▶ Only pick up if you can carry more.
    ▶ Only drop if you are carrying something.
    ▶ Stay where you are when picking up and dropping.
    ▶ Pick up 'meal' in 'kitchen'.
    ▼ Deliver 'meal' to 'r1'.
        ([]((next(s.mem_deliver_r1) <-> (s.mem_deliver_r1 | (next(s.r1) & next(s.drop))))))
        ([]<>(s.mem_deliver_r1))
    ▶ Deliver 'meal' to 'r2'.
    ▶ Deliver 'meal' to 'r3'.
    ▶ Deliver 'meal' to 'r4'.
    ▶ Deliver 'meal' to 'r5'.
    ▶ Deliver 'meal' to 'r6'.
▼ Don't go into any public rooms.
  ▼ Action: do not 'go', Location: 'rooms'
    ▼ Do not go to 'hall_c'.
        ([](!s.hall_c))
    ▶ The robot does not begin in 'hall_c'.
    ▶ Do not go to 'lounge'.
    ▶ The robot does not begin in 'lounge'.
```

Fig. 6 Screenshot of feedback for Specification 4

By simply removing the restriction in Line 2 (or, alternatively, adding an assumption that the target will never enter the kitchen) the specification can be made realizable. Future work will automate the suggestion of such assumptions that would make the specification realizable.

5.5 Reporting current behavior

Figure 8 shows examples of the robot responding to queries regarding its current action. The specification being executed is only the command *Follow me,* a modified version of



	alarm	sbit0	sbit1	sbit2	sbit3	Region
1	False	False	True	True	True	p2 (r5)
2	False	False	True	True	True	p9 (hall_w)

Your goal:
Follow the target to 'kitchen'. (Because you said "Follow me.")
Current environment state:

sbit1	sbit0	sbit3	sbit2	alarm

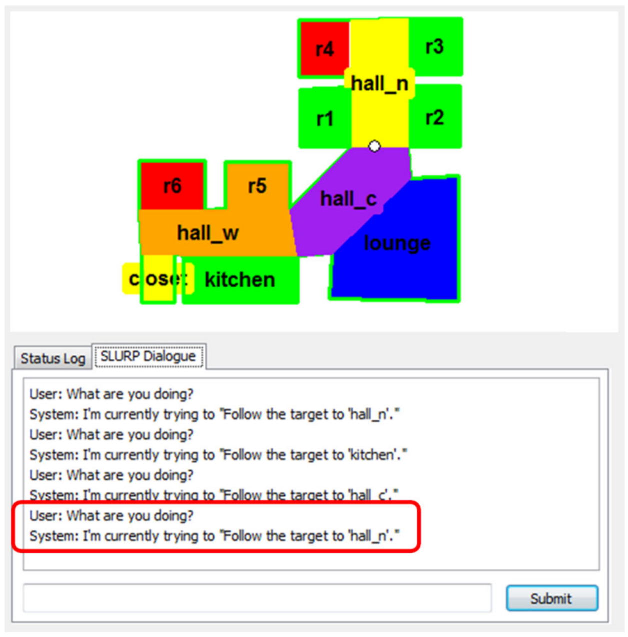
Please choose your response:
Move to region p8 (kitchen)
Actuator states:
Internal propositions:

Invalid move because it violates: 'Do not go to kitchen. (Because you said "Don\'t go into the kitchen.")'

Execute Move >>

Currently in step #3

Fig. 7 Screenshot of interactive visualization tool for Specification 5. The user is prevented from following the target into the kitchen in the next step (denoted by the *blacked out* region) due to the portion of the specification displayed



User: What are you doing?
System: I'm currently trying to "Follow the target to 'hall_n'."
User: What are you doing?
System: I'm currently trying to "Follow the target to 'kitchen'."
User: What are you doing?
System: I'm currently trying to "Follow the target to 'hall_c'."
User: What are you doing?
System: I'm currently trying to "Follow the target to 'hall_n'."

Fig. 8 Screenshot of robot response to query: *What are you doing?*

Specification 5 where the robot may follow the user without restriction. The robot is asked what it is doing each time the user moves. As the command is expanded into a separate following goal for each room, the robot is able to report what it is doing in more specific language than the original specification.

Note that this work identifies *logical* causes of unsatisfiability and unrealizability in the specification, and does not explain unsynthesizability resulting from the dynamics of the vehicle. Extending the described techniques to identify causes of unsynthesizability that relate to the availability of continuous controllers has been identified as a potential direction of future research.

6 Conclusion

This paper presents an integrated system that allows non-expert users to control robots performing high-level, reactive tasks using a natural language interface. The depth of integration between the natural language components and the synthesis, unsynthesizable core-finding, and execution modules allows for natural language specifications, feedback on specification errors, and explanation of current goals during execution.

The described user study demonstrates that the approach taken for natural language understanding, combining standard parsing and tagging modules with a deterministic semantic parsing system, is capable of understanding non-expert user commands with high accuracy. As shown in the error analysis, the greatest improvements to the performance of the system would come from training the tagger on more data containing imperatives and expanding the vocabulary coverage of the system to cover more of the verbs used by users. The design of the system makes adding support for additional verbs straightforward as there is no grammar to update, only a vocabulary list to amend.

While the use of domain-general language processing tools allows for an extensible system, some constructs common in the robotics domain may not be understood correctly. For example, a user might want to use the phrase *turn on your camera* instead of *activate your camera* as in the examples in this paper. Unfortunately, particles such as *on* are often assigned incorrect part-of-speech tags, resulting in failure to understand commands such as *turn on* and *turn off*. The collection of a corpus of robot control interactions for use in training broad-domain language models for robot control would result in higher performance from the natural language processing and semantic extraction components.

The example of a robot in a hospital setting shows that even in simple specifications, users can accidentally create complex synthesizability issues. However, these issues can be debugged efficiently using a minimal core described in natural language. The unsynthesizability feedback and explanation of current goals depend on the novel generation tree data structure introduced in this work, which takes a normally opaque LTL generation process and makes it visible to the user.

There are significant challenges in designing a robust mapping from natural language semantics to LTL formulas. While simple motion commands and actuations can be straightforwardly mapped into logical form, more complex actions like the delivery of objects can be more difficult to translate. However, the benefit of the proposed system is that the effort is invested just once in the design of the mapping, not repeatedly for each specification as it would be for users writing specifications in LTL or structured language. The system presented is easily applied to scenarios where the specification is largely centered around motion and simple actuations, but can be extended by users proficient in LTL to more complex scenarios by adding support for additional behaviors. Future work might explore the automatic learning of mappings between semantic structures and LTL representations. Learning a mapping that allows for complex specifications to be reliably synthesized would require the system to learn many of the subtleties of LTL specification authoring, such as maintaining consistent tense across the mapping for each type of action.

A number of issues in the LTL generation process merit further consideration. While this paper discusses some of the challenges regarding the application of negation, further work should address more formal paradigms for mapping actions to LTL formulas. The production of an ontology of common actions and the type of formulas that they produce—for example, safety conditions, adding goals, constraining the initial state—in their negated and positive forms would be a step toward a more general solution to the problem of mapping natural language to LTL. Previous work has relied heavily on grammar formalisms to ease semantic extraction. While those formalisms provide a structure for easy extraction of semantic roles, they are not robust to natural input and do not address the more urgent problem of robust logical representations over large sets of possible actions which remain appropriately synthesizable in complicated specifications.

The examples presented here have focused on the execution of a single specification. However, it is possible that over the course of a mission requirements may change and new commands may be given. While the underlying execution environment supports resynthesis and transitioning execution to a new automaton during execution, this introduces a number of challenges in the communication between system and user. Future work should explore means for maintaining the level of natural language integration presented in this paper across more complex execution paradigms, handling events such as changes in the workspace topology, as would occur when operating in environments that are only partially known.

Acknowledgments We would like to thank Taylor Turpen, Israel Geselowitz, and Kenton Lee for their assistance with software development and data collection. This work was supported in part by: ARO MURI (SUBTLE) W911NF-07-1-0216, NSF CAREER CNS-0953365, DARPA N66001-12-1-4250, and TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

Berant, J., & Liang, P. (2014). Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1415–1425).

Bhatia, A., Kavraki, L. E., & Vardi, M.Y. (2010). Sampling-based motion planning with temporal goals. In *IEEE International Conference on Robotics and Automation (ICRA), IEEE* (pp. 2689–2696).

Biere, A. (2008). PicoSAT essentials. *Journal on Satisfiability Boolean Modeling and Computation (JSAT)*, *4*, 75–97.

Bikel, D. M. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*, *30*(4), 479–511.

Bobadilla, L., Sanchez, O., Czarnowski, J., Gossman, K., & LaValle, S. (2011). Controlling wild bodies using linear temporal logic. In *Robotics: Science and Systems (RSS)*.

Brooks, D., Lignos, C., Finucane, C., Medvedev, M., Perera, I., Raman, V., Kress-Gazit, H., Marcus, M., & Yanco, H. (2012). Make it so: Continuous, flexible natural language interaction with an autonomous robot. In *Proceedings of the Grounding Language for Physical Systems Workshop at the 76th AAAI Conference on Artificial Intelligence*.

Chen, D. L., & Mooney, R.J. (2011). Learning to interpret natural language navigation instructions from observations. In *Proceedings of the 25th AAAI Conference on Artifical Intelligence* (pp. 859–865).

Cizelj, I., & Belta, C. (2013). Negotiating the probabilistic satisfaction of temporal logic motion specifications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 4320–4325).

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge, MA: MIT Press.

Dzifcak, J., Scheutz, M., Baral, C., & Schermerhorn, P. (2009). What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *IEEE International Conference on Robotics and Automation (ICRA)* (pp. 4163–4168).

Fainekos, G. E. (2011). Revising temporal logic specifications for motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)* (pp. 40–45).

Finucane, C., Jing, G., & Kress-Gazit, H. (2010). LTLMoP: Experimenting with language, temporal logic and robot control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 1988–1993).

Gabbard, R., Marcus, M., & Kulick, S. (2006). Fully parsing the Penn Treebank. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (NAACL HLT)* (pp. 184–191).

Karaman, Frazzoli. (2009). Sampling-based motion planning with deterministic μ-calculus specifications. In *IEEE Conference on Decision and Control (CDC)* (pp. 2222–2229).

Kim, K., Fainekos, G. E., & Sankaranarayanan, S. (2012). On the revision problem of specification automata. In *IEEE International Conference on Robotics and Automation (ICRA)* (pp. 5171–5176).

Kloetzer, M., & Belta, C. (2008). A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, *53*(1), 287–297.

Kress-Gazit, H., Fainekos, G. E., & Pappas, G. J. (2008). Translating structured english to robot controllers. *Advanced Robotics*, *22*(12), 1343–1359.

Kress-Gazit, H., Fainekos, G. E., & Pappas, G. J. (2009). Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, *25*(6), 1370–1381.

Matuszek, C., Fox, D., & Koscher, K. (2010). Following directions using statistical machine translation. In *Human-Robot Interaction (HRI)* (pp. 251–258).

Matuszek, C., FitzGerald, N., Zettlemoyer, L., Bo, L., & Fox, D. (2012). A joint model of language and perception for grounded attribute learning. In *Proceedings of the 29th International Conference on Machine Learning (ICML)* (pp. 1671–1678).

Matuszek, C., Herbst, E., Zettlemoyer, L., & Fox, D. (2013). Learning to parse natural language commands to a robot control system. *Experimental Robotics*, *88*, 403–415.

Piterman, N., Pnueli, A., & Sa'ar, Y. (2006). Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (pp. 364–380).

Poon, H., & Domingos, P. (2009). Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1–10).

Raman, V., & Kress-Gazit, H. (2011). Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP. In *Computer Aided Verification (CAV)* (pp. 663–668).

Raman, V., & Kress-Gazit, H. (2013a). Explaining impossible high-level robot behaviors. *IEEE Transactions on Robotics*, *29*, 94–104.

Raman, V., & Kress-Gazit, H. (2013b). Towards minimal explanations of unsynthesizability for high-level robot behaviors. In

IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 757–762).

Raman, V., & Kress-Gazit, H. (2014). Unsynthesizable cores: Minimal explanations for unsynthesizable high-level robot behaviors. arXiv:1409.1455.

Raman, V., Lignos, C., Finucane, C., Lee, KCT., Marcus, M., & Kress-Gazit, H. (2013). Sorry Dave, I'm afraid I can't do that: Explaining unachievable robot tasks using natural language. In *Robotics: Science and Systems (RSS)*.

Schuler, K. (2005). Verbnet: A broad-coverage, comprehensive verb lexicon. PhD thesis, University of Pennsylvania.

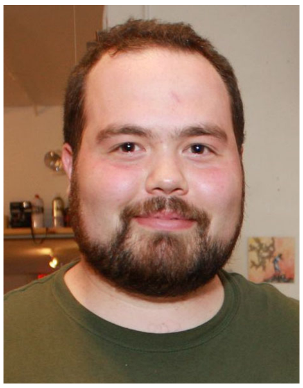
Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A. G., Teller, S. J., & Roy, N. (2011). Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the 25th AAAI Conference on Artifical Intelligence* (pp. 1507–1514).

Toutanova, K., Klein, D., Manning, C. D., & Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL HLT) - Volume 1* (pp. 173–180).

Wongpiromsarn, T., Topcu, U., & Murray, R. M. (2010). Receding horizon control for temporal logic specifications. In *Hybrid Systems: Computation and Control (HSCC)* (pp. 101–110).

Constantine Lignos is a postdoctoral fellow at The Children's Hospital of Philadelphia. His research interests include computational models of language acquisition and processing, unsupervised learning of linguistic structure, and natural language understanding and dialog for autonomous systems. He earned a Ph.D. in Computer and Information Science in 2013 from the University of Pennsylvania and a BA from Yale University in 2006. He worked on automotive dialog systems at Microsoft from 2006 to 2008.

Vasumathi Raman is a postdoctoral scholar in the Department of Computing and Mathematical Sciences at the California Institute of Technology, Pasadena, CA. Her research interests lie at the intersection of autonomous high-level control and formal methods, with applications in robotics and the control of cyber-physical systems. She earned a Ph.D. in Computer Science in 2013 from Cornell University, NY, and a BA from Wellesley College, MA.



Cameron Finucane is a research assistant in the Autonomous Systems Lab at Cornell University. As a member of Professor Hadas Kress-Gazit's research group, his work focuses on high-level robot control and the development of the LTLMoP toolkit. He previously studied electrical engineering and linguistics at the University of Pennsylvania and at Waseda University.

Mitchell Marcus is the RCA Professor of Artificial Intelligence in the Department of Computer and Information Science at the University of Pennsylvania, where he is also Professor of Linguistics. His research interests include statistical natural language processing, human-robot communication, and cognitively plausible models for automatic acquisition of linguistic structure. He was the principal investigator for the Penn Treebank Project, part of the OntoNotes text annotation effort within the DARPA GALE project, and was recently the PI on an ARO funded MURI project investigating natural language interfaces for autonomous robots. He has served as Chair of the Department of Computer and Information Science at Penn, as well as President of the Association for Computational Linguistics. He is a fellow of the American Association of Artificial Intelligence, and a founding fellow of the Association for Computational Linguistics and is chair of the advisory board of the Center of Excellence in Human Language Technologies at Johns Hopkins University. He received his Ph.D. in 1978 from the MIT Artificial Intelligence Lab, and was a Member of Technical Staff at AT&T Bell Laboratories before coming to the University of Pennsylvania in 1987.

Hadas Kress-Gazit received her Ph.D. in Electrical and Systems Engineering from the University of Pennsylvania in 2008. She is currently an Assistant Professor at the Sibley School of Mechanical and Aerospace Engineering at Cornell University. Her research focuses on creating verifiable robot controllers for complex high-level tasks using logic, verification methods, synthesis, hybrid systems theory and computational linguistics. She is a recipient of the NSF CAREER award in 2010 and the DARPA Young Faculty Award in 2012.