Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language

Vasumathi Raman*, Constantine Lignos†, Cameron Finucane‡, Kenton C.T. Lee†, Mitch Marcus†
and Hadas Kress-Gazit‡
*Department of Computer Science, Cornell University, Ithaca, NY 14850
Email: vraman@cs.cornell.edu
†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104
Email: lignos@cis.upenn.edu, kentonl@seas.upenn.edu, mitch@cis.upenn.edu
‡Sibley School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY 14850
Email: cpf37@cornell.edu, hadaskg@cornell.edu

***Abstract*—This paper addresses the challenge of enabling non-expert users to command robots to perform complex high-level tasks using natural language. It describes an integrated system that combines the power of formal methods with the accessibility of natural language, providing correct-by-construction controllers for high-level specifications that can be implemented, and easy-to-understand feedback to the user on those that cannot be achieved. This is among the first works to close this feedback loop, enabling users to interact with the robot in order to identify a succinct cause of failure and obtain the desired controller. The supported language and logical capabilities are illustrated using examples involving a robot assistant in a hospital.**

I. Introduction

As robots become more ubiquitous, multi-capable and general-purpose, it is desirable for them to be easily controllable by non-expert users. The near future will likely see robots in homes and offices, performing everyday tasks such as fetching coffee and tidying rooms. 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 to robot control have enabled automated synthesis of correct-by-construction hybrid controllers for complex high-level tasks (e.g., [15, 13, 1, 4, 17, 26]).

However, most current approaches require the user to provide task specifications in logic, or a similarly structured specification language [16]; users must formally reason about system requirements rather than providing an intuitive description of the desired outcome. This motivates a robot control platform that allows users to specify behaviors via natural language (text or speech) and provides either the desired robot controller or an explanation for why one cannot exist.

This paper presents an integrated platform that enables robot control from natural language specifications for situated high-level tasks. Natural language commands to the robot are parsed using semantic analysis into a formal specification which is used to synthesize a hybrid controller. If no implementation exists, the user is provided with an explanation and the portions of the specification that cause failure. The proposed system thus 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.

There are several previous approaches that use natural language for controlling robots. Some frameworks translate instructions in unconstrained natural language into formal goal descriptions and action scripts for tasks like navigation and manipulation [9, 18, 25]. Others map high-level instructions to more fine-grained sequences of commands, filling in missing information [5]. This paper builds on components presented in [6], adding natural language capabilities to deal with a wider class of specifications, and providing fine-grained feedback on specifications that cannot be implemented.

On the controller synthesis front, recent work has tackled the problem of analyzing high-level specifications that are unsynthesizable. Feedback about the cause of unsynthesizability can be provided to the user in the form of a modified specification [10, 14], a highlighted fragment of the original specification [20], or by allowing the user to interact with an adversarial environment that prevents the robot from achieving the specified behavior [21, 22]. Building upon these approaches, this work provides minimal explanations of unsatisfiable specifications, as described in Section V, and provides fine-grained natural language feedback that is not necessarily just a subset of the original natural language specification. It also enhances the interactive visualization tool with feedback on why particular robot actions may be disallowed in a given state, as described in Section VI-B.

II. System Overview

Fig. 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; these are described in detail in Section III. This module is the natural language connection between the user and the logical representations used within the Linear Temporal Logic MissiOn Planning (*LTLMoP*) toolkit [11]. LTLMoP, which also interfaces with the SAT solver PicoSAT [2], provides an environment for creating, analyzing, and executing specifications.

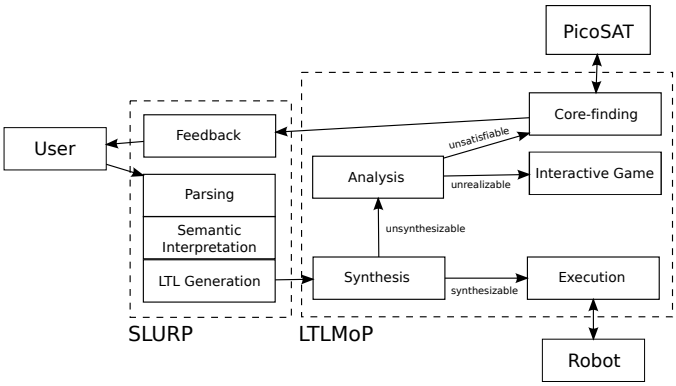


Fig. 1: System Overview

Subcomponents of LTLMoP are detailed in Section IV.

III. Transforming Natural Language into Logic with SLURP

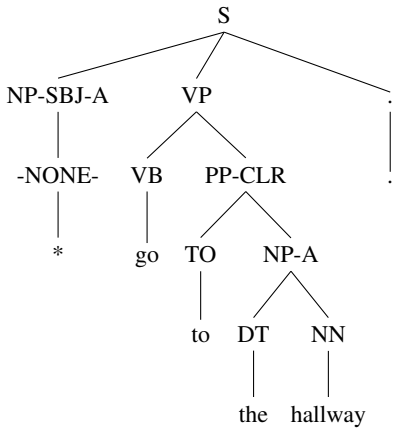
The Situated Language Understanding Robot Platform (*SLURP*) enables the conversion of natural language specifications into Linear Temporal Logic formulas. The user's instructions are processed through a pipeline of natural language components which identify the syntactic structure of the sentences, 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 allow the unification of semantic information and natural language representations [8], this work uses a combination of robust, general-purpose components. 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.

A. Identifying Linguistic Structure

Before a sentence may be converted into logical formulas, the linguistic structure of the sentence must be identified.

1) Parsing: 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 the pipeline of natural language processing components used by Brooks et al. [6]: the Bikel parser [3] combined with the null element (understood subject) restoration of Gabbard et al. [12] to parse sentences. Before being given to the parser, the input is tagged using MXPOST [23]. The models used by these systems require no in-domain training. The output of these modules is given in Figure 2A.

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*.



(A) Tagging, parsing, and null element restoration

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

(B) 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:
$\Box\Diamond(s.mem\_visit\_hallway)$

(C) LTL formula generation

Fig. 2: Conversion of the sentence "Go to the hallway." into LTL formulas through parsing, semantic interpretation, and LTL generation.

To extract verbs and their arguments from parse trees, SLURP uses VerbNet [24], 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 general sense *CARRY*, because in some contexts they are roughly equivalent in meaning. For each sense, VerbNet provides a set of frames, which indicates the possible arguments to the sense. Consider the following sentence: *Carry meals to all patient rooms*. The verb *carry* is mapped to the sense *CARRY*. An example of a frame for this sense is [*AGENT, VERB, THEME, TO TOWARDS, DESTINATION*]. Each role in the frame, subject to its associated syntactic constraints, is mapped to a part of the parse tree. In this case, SLURP creates the following mapping: [*AGENT* → *, *VERB* → *carry, ..., DESTINATION* → *all patient rooms*]. Among the frames that completely match

the parse tree, SLURP chooses the frame that expresses the most semantic roles. The chosen match is then used to fill in the appropriate fields in the command.

B. Generation of LTL Formulas

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

1) Linear Temporal Logic (LTL): 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. 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. [7].

Task specifications in this work are expressed as LTL formulas of 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 for the environment (e) and the robot (s). An overview of the process that generates LTL formulas from natural language is given in Figure 2. First, the linguistic structure of the sentence is recovered, then semantic information is extracted from the structure, and finally the semantic information is used to generate LTL formulas.

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. SLURP can easily be extended to cover additional actions, but those covered so far by the system are:

1) Going to rooms (*Go to the kitchen.*)
2) Patrolling (continuously visiting) rooms (*Patrol the hallway.*)
3) Never going to a room (*Don't go to the lounge.*)
4) Searching a room (*Search the cafeteria.*)
5) Following (*Follow me.*)
6) Enabling/disabling actuators (*Activate your camera.*)
7) 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 expressed using LTL. Each command may be freely combined with conditional structures (e.g., "If you hear an alarm..."), negation, coordination, and quantification.

3) Quantification: To allow the same command to be conveniently performed on a set of locations, commands can be specified over quantified arguments. For example, the command "go to all patient rooms" will be unrolled to apply to all rooms tagged with the keyword "patient."

4) LTL Generation: For each supported command, LTL is generated by macros which create the appropriate assumptions, restrictions, and goals. In the example given in Figure 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 goals ($goal(x)$ generates $\Box\Diamond(x)$), persistent memories ($memory(x)$ generates $\Box(\bigcirc s.mem\_x \Leftrightarrow (s.mem\_x \vee \bigcirc s.x))$), and complete at least once ($alo(x)$ generates $(goal(s.mem\_x) \wedge memory(x))$). For example, patrolling a room is mapped to $goal(room)$, and going to a room is mapped to $alo(room)$. More complex examples of generation involving combinations of these macros are given in Sections VI and VII. A challenge in creating a correct mapping is that negation of a command does not necessarily imply its logical negation; a command to avoid a room (or equivalently not go to it) simply generates $\Box \neg s.room$ without a memory as in the non-negated case.

5) The 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 Figure 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, with LTL formulas as leaves. The intermediate nodes are 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 Figure 3, if the current goal being pursued during execution is $\Box\Diamond s.mem\_visit\_lounge$, the system responds: "I'm currently trying to 'visit lounge'." In cases where the original instruction involves quantification, identification of the sub-goal 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.

[1]This arguably unintuitive translation is due to specifications in LTLMoP being restricted to a fragment of LTL for computational reasons [17].

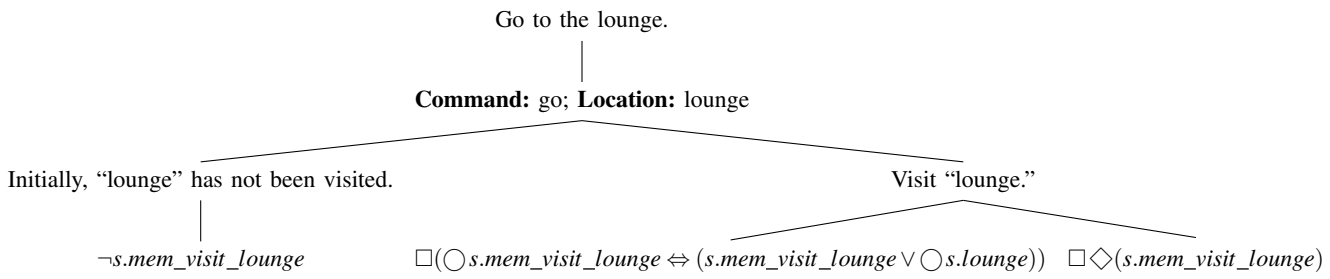


Fig. 3: Generation tree for "Go to the lounge."

IV. From Linear Temporal Logic to Continuous Control with LTLMoP

This section discusses the construction of provably-correct controllers from LTL specifications using the Linear Temporal Logic MissiOn Planning (LTLMoP) toolkit.

A. 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. [19] 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. Details of the synthesis process and the resulting hybrid controller can be found in Kress-Gazit et al. [17], Finucane et al. [11].

B. Specification Analysis

If an implementing automaton exists for a given LTL formula, it is called *realizable*. Otherwise, the algorithm presented in Raman and Kress-Gazit [22] 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. However, the granularity of the feedback provided by this algorithm is relatively coarse. For example, it identifies a contradiction within the system safety conditions, but cannot pinpoint the exact safeties that are contradicting. Section V describes algorithms for reducing the problem to a minimal, *core* subset of the original specification.

Given a set of LTL subformulas that cause the specification to be unsynthesizable, 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 [11], this is done by highlighting the sentences that produced the corresponding LTL [20]. The generation of natural language feedback from the identified portions of the LTL formula in SLURP is discussed in Section VI.

V. Identifying Minimal Unsatisfiable Cores in LTL Specifications

As mentioned in Section IV-B, when a specification does not yield an implementing automaton, existing algorithms provide feedback of a coarse granularity [22]. This section describes techniques implemented in this work that provide fine-grained feedback in certain cases by tracing the problem back to a minimal explanation, and identifies the challenges that must be overcome to extend these techniques to the remaining cases.

When controller synthesis fails the specification is called *unsynthesizable*. 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 [22].

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 able to change its state infinitely often 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. Livelock occurs when there is no deadlock, but one or more goals cannot be reached while still following the robot safety conditions.

Previous work produced explanations of unsynthesizability in terms of combinations of the specification components (i.e., initial, safety and liveness conditions). However the true conflict often lies in small subformulas of these components.

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)

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. However, the algorithm presented by Raman and Kress-Gazit [22] will return the entirely of φ_s^t along with the goal of

visiting the kitchen, declaring that this goal is in conflict with some subset of the safeties (but not identifying the exact subset). 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.

As part of the system presented in this paper, unsatisfiable components of the robot specification φ_s are further analysed to narrow down the cause of unsatisfiability for both deadlock and livelock. Extending these techniques to the environment assumptions φ_e is straightforward.

A. Unsatisfiable Cores for Deadlock

Given a depth d and an LTL safety formula φ over propositions $p \in AP$, there exists a propositional formula ψ over $\bigcup_{1\leq i\leq d+1} AP^i$, where $p^i \in AP^i$ represents the value of $p \in AP$ at time step i, constructed as:

$$\psi^d(\varphi) = \bigwedge_{1\leq i\leq d} \varphi[\bigcirc p/p^{i+1}][p/p^i],$$

where $\varphi[a/b]$ represents φ with all occurrences of subformula a replaced with b. Formula $\psi^d(\varphi)$ is called the depth-d *unrolling* of φ. Consider Specification 2. $\psi^1(\varphi_s^t) = \neg s.kitchen^1 \wedge s.camera^2$ and $\psi^2(\varphi_s^t) = \neg s.kitchen^1 \wedge s.camera^2 \wedge \neg s.kitchen^2 \wedge s.camera^3$, where proposition $s.kitchen^i$ represents the truth value of $s.kitchen$ at time step i.

When deadlock is identified [21, 22], a minimal unsatisfiable core is produced by incrementally creating unrollings of the robot safety formula φ_s^t, until a depth d is reached where $\psi^d_{fromInit} = \varphi_s^i[p/p^1] \wedge \psi^d(\varphi_s^t)$ is unsatisfiable. In other words, there is no valid sequence of actions that follow the safety to d time steps starting from the initial condition. An off-the-shelf satisfiability (SAT) solver can be used to determine unsatisfiability of $\psi^d_{fromInit}$; this work uses PicoSAT [2]. If $\psi^d_{fromInit}$ is unsatisfiable, the same SAT solver yields a minimal unsatisfiable subformula, which is then mapped back to the originating portions of the safety and initial formulas.

Specification 2 Core-finding example – unsatisfiable deadlock

1) Start in the kitchen (φ_s^i):
   $s.kitchen$
2) Avoid the kitchen (φ_s^i, φ_s^t):
   $\neg s.kitchen \wedge \Box \neg s.kitchen$
3) Always activate your camera (φ_s^t):
   $\Box \bigcirc s.camera$

In Specification 2, a deadlocked specification, the described method begins at the initial state described by φ_s^i (lines 1 and 2), and unrolls it to $\psi^1_{fromInit} = s.kitchen^1 \wedge \neg s.kitchen^1 \wedge \neg s.kitchen^1 \wedge s.camera^2$ above. Note that $\varphi^1_{fromInit}$ is already unsatisfiable, and the core is given by the subformula $s.kitchen^1 \wedge \neg s.kitchen^1$, which in turn maps back to lines 1 and 2. This is because the two statements combined require the robot to both start in the kitchen and not start in the kitchen. Section VII contains another example demonstrating unsatisfiable core-finding for deadlocked specifications.

B. Unsatisfiable Cores for Livelock

In the case of livelock, a similar unrolling procedure can be applied to determine the core set of clauses that prevent a goal from being fulfilled. A propositional formula is generated by unrolling the robot safety from the initial state for a predetermined number of time steps, with an additional clause representing the goal being required to hold at the last time step. Consider the livelocked Specification 3, referring to a robot operating in the workspace depicted in Fig. 4.

Specification 3 Core-finding example – unsatisfiable livelock

1) Start in the kitchen (φ_s^i):
   $s.kitchen$
2) Avoid hall_w (φ_s^i, φ_s^t):
   $\neg s.hall\_w \wedge \Box \neg s.hall\_w$
3) Always activate your camera (φ_s^t):
   $\Box \bigcirc s.camera$
4) Patrol r3 (φ_s^g):
   $\Box \Diamond s.r3$

Unrolling the robot safety to depth 5 results in:

$$\psi^5_{fromInit} = s.kitchen^1 \wedge \bigwedge_{1\leq i\leq 5} \neg s.hall\_w^i \wedge s.r3^5 \wedge \bigwedge_{2\leq i\leq 5} s.camera^i \wedge \bigwedge_{1\leq i\leq 5} \varphi^i_{topo}$$

where φ^i_{topo} represents the topology constraints on the robot at time i. Note that $\psi^5_{fromInit}$ is unsatisfiable, and the core is given by the subformula:

$$s.kitchen^1 \wedge \bigwedge_{1\leq i\leq 5} \neg s.hall\_w^i \wedge s.r3^5 \wedge \bigwedge_{1\leq i\leq 5} \varphi^i_{topo}$$

which maps back to (1), (2) and (4). This is because the robot cannot reach $r3$ without passing through *hall_w*. Section VII contains another example demonstrating unsatisfiable core-finding for livelocked specifications.

In the case of deadlock, the propositional formula can be built one step at a time until it is found to be unsatisfiable. This gives us a sound and complete method for determining the depth to which the safety must be unrolled in order to identify an unsatisfiable core for deadlock. For livelock, on the other hand, determining the shortest depth that will produce a meaningful core is a much bigger challenge. Consider the above example. For unroll depths less than or equal to 3, the unsatisfiable core returned will include just the workspace topology, since the robot cannot reach r3 from the kitchen in 3 steps or fewer, even if it is allowed into *hall_w*; however, this is not a meaningful core. Determining the shortest depth that will produce a meaningful core is a future research challenge, and for the purpose of this work, a fixed depth of 15 time steps was used for the examples presented.

C. Identifying the Cause of Unrealizability

If the specification is unrealizable rather than unsatisfiable, the above techniques do not apply directly to identify a core, since all environment strategies must be considered when unrolling the system safety relations, resulting in not one but

	Deadlock	Livelock
Unsatisfiable	Core (minimal subset of safeties)	Core (minimal subset of safeties, single goal)
Unrealizable	Entire safety formula; interactive game	Entire safety formula, single goal; interactive game

TABLE I: Summary of the types of feedback provided

exponentially many possible propositional formulas at each time step. It is unclear what an unrealizable "core" would be in this context, and future work will develop methods of narrowing down the cause of unrealizability to a suitably-defined core. At present, the feedback provided in the unrealizable case is at the same granularity as that of Raman and Kress-Gazit [21], Raman and Kress-Gazit [22]. However, unsatisfiable cores enable a useful enhancement to the interactive visualization game, as described in Section VI-B.

Table I summarizes the feedback provided in cases of unrealizability and unsatisfiability, for both deadlock and livelock.

VI. Providing Users with Feedback on Specifications

Once the guilty portions of the LTL formula generated from a task specification have been identified, the cause of failure must be presented to the user. This section describes two modes of communicating the cause of unsynthesizability deployed in the presented system.

A. 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:

$$\begin{aligned}&\square(\neg s.kitchen)\\&\square(\bigcirc s.mem\_visit\_kitchen \Leftrightarrow\\&\quad(s.mem\_visit\_kitchen \vee \bigcirc s.kitchen))\\&\square\diamondsuit(s.mem\_visit\_kitchen)\end{aligned}$$

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 Section VII.

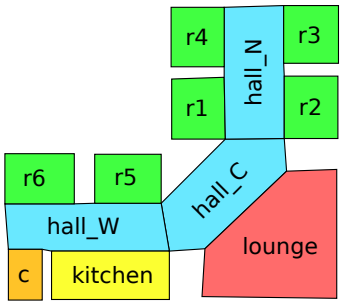


Fig. 4: Map of hospital workspace ("c" is the closet)

B. 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 (shown in Fig. 6) to demonstrate an environment behavior that will cause the robot to fail. This 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. An example of this tool in use is given in Section VII-B.

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.

VII. Hospital Example

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 hospital (a map of the workspace is shown in Fig. 4). The robot is able to detect the location of the user, record video with its camera, and pick up and deliver objects.

A. Unsatisfiability

Specification 4 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.

1) Deadlock: Unsatisfiable deadlock can arise when a robot safety constraints are in direct conflict with one another. For example, in Specification 4, 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

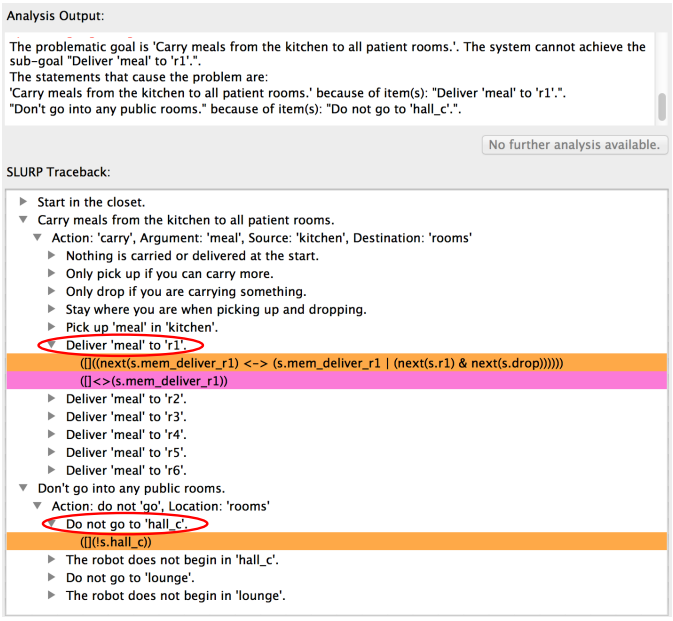


Fig. 5: Screenshot of feedback for Specification 5.

core-finding component correctly narrows down the problem as follows:

> 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'.".

We notice that, for example, 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 5 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.

2) Livelock: Unsatisfiable livelock is exhibited by the meal delivery mission shown in Specification 5, 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. 5). 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.

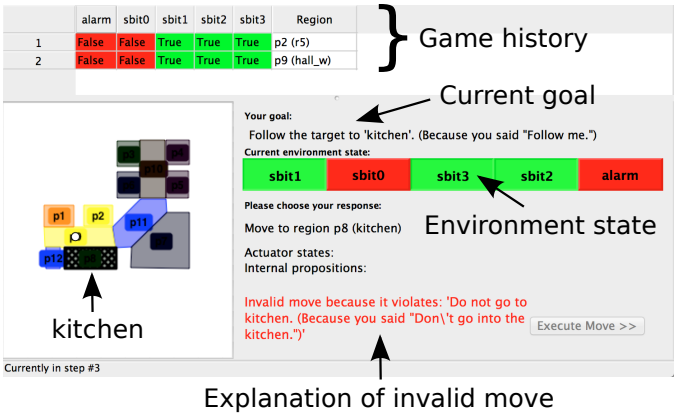


Fig. 6: Screenshot of interactive visualization tool for Specification 6. The user is prevented from following the target into the kitchen in the next step (denoted by the blacked out region) due to the portion of the specification displayed.

B. Unrealizability

As introduced in Section VI-B, unrealizable specifications can be analyzed using an interactive visualization tool (see Fig. 6). For example, in the case of Specification 6, 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 6 Example of unrealizability

1) Follow me.
2) Avoid the kitchen.

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.

VIII. Conclusions

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 components of the proposed system are discussed in detail, and their contributions to the system's capabilities illustrated with examples involving a robot assistant in a hospital. Informal instructions to the robot are transformed into formal specifications, and used to synthesize a hybrid controller when possible. For unimplementable specifications, fine-grained analysis provides the user with a concise explanation of the portions of the specification that cause the failure; this is the first work that provides the user with natural language feedback on failed specifications. Future work will extend the core-finding capabilities from unsatisfiable to unrealizable specifications,

and extend the natural language parsing and semantic analysis to wider classes of specifications by considering a variety of contexts for high-level robot control.

Acknowledgments

This work was supported by NSF CAREER CNS-0953365, ARO MURI (SUBTLE) W911NF-07-1-0216, NSF ExCAPE and DARPA N66001-12-1-4250.

References

[1] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Sampling-based motion planning with temporal goals. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2689–2696. IEEE, 2010.

[2] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[3] Daniel M. Bikel. Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4):479–511, 2004.

[4] Leonardo Bobadilla, Oscar Sanchez, Justin Czarnowski, Katrina Gossman, and Steven LaValle. Controlling wild bodies using linear temporal logic. In *Robotics: Science and Systems (RSS)*, Los Angeles, CA, USA, June 2011.

[5] S. R. K. Branavan, Luke S. Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1268–1277, 2010.

[6] Daniel Brooks, Constantine Lignos, Cameron Finucane, Mikhail Medvedev, Ian Perera, Vasumathi Raman, Hadas Kress-Gazit, Mitch Marcus, and Holly Yanco. Make it so: Continuous, flexible natural language interaction with an autonomous robot. In *Grounding Language for Physical Systems Workshop at the AAAI Conference on Artificial Intelligence*, 2012.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. ISBN 0262032708.

[8] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul Schermerhorn. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *ICRA*, Kobe, Japan, May 2009.

[9] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul W. Schermerhorn. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *ICRA*, pages 4163–4168, 2009.

[10] Georgios E. Fainekos. Revising temporal logic specifications for motion planning. In *ICRA*, pages 40–45, 2011.

[11] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *IEEE/RSJ International. Conf. on Intelligent Robots and Systems (IROS)*, pages 1988 – 1993, 2010.

[12] Ryan Gabbard, Mitch Marcus, and Seth Kulick. Fully parsing the Penn Treebank. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics (NAACL HLT)*, pages 184–191. Association for Computational Linguistics, 2006.

[13] Karaman and Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *IEEE Conference on Decision and Control (CDC)*, Shanghai, China, December 2009.

[14] Kangjin Kim, Georgios E. Fainekos, and Sriram Sankaranarayanan. On the revision problem of specification automata. In *ICRA*, pages 5171–5176, 2012.

[15] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transaction on Automatic Control*, 53(1):287–297, 2008.

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

[17] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6): 1370–1381, 2009.

[18] Cynthia Matuszek, Dieter Fox, and Karl Koscher. Following directions using statistical machine translation. In *Human-Robot Interaction (HRI)*, pages 251–258, 2010.

[19] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 06)*, pages 364–380. Springer, 2006.

[20] Vasumathi Raman and Hadas Kress-Gazit. Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP. In *Computer Aided Verification (CAV)*, pages 663–668, 2011.

[21] Vasumathi Raman and Hadas Kress-Gazit. Automated feedback for unachievable high-level robot behaviors. In *ICRA*, pages 5156–5162, 2012.

[22] Vasumathi Raman and Hadas Kress-Gazit. Explaining impossible high-level robot behaviors. *Robotics, IEEE Transactions on*, PP(99):1 –11, 2012. ISSN 1552-3098. doi: 10.1109/TRO.2012.2214558.

[23] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Empirical Methods in Natural Language Processing (EMNLP)*, volume 1, pages 133–142, 1996.

[24] K.K. Schuler. *VerbNet: A broad-coverage, comprehensive verb lexicon*. PhD thesis, University of Pennsylvania, 2005.

[25] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R. Walter, Ashis Gopal Banerjee, Seth J. Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011.

[26] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Hybrid Systems: Computation and Control (HSCC)*, pages 101–110, 2010.