Explaining Impossible High-Level Robot Behaviors

Vasumathi Raman, Hadas Kress-Gazit

***Abstract*—A key challenge in robotics is the generation of controllers for autonomous, high-level robot behaviors comprising non-trivial sequences of actions including reactive and repeated tasks. When constructing controllers to fulfil such tasks, it is often not known a priori whether the intended behavior is even feasible; plans are modified on the fly to deal with failures that occur during execution, often still without guaranteeing correct behavior. Recently, formal methods have emerged as a powerful tool for automatically generating autonomous robot controllers that guarantee desired behaviors expressed by a class of temporal logic specifications. However, when the specification cannot be fulfilled, these approaches do not provide the user with a source of failure, making the troubleshooting of specifications an unstructured and time-consuming process. This paper describes an algorithm for automatically analyzing an unsynthesizable specification in order to identify causes of failure. It also introduces an interactive game for exploring possible causes of unsynthesizability, in which the user attempts to fulfill the robot specification against an adversarial environment. The proposed algorithm and game are implemented as features within the LTLMoP toolkit for robot mission planning.**

***Index Terms*—IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

THE goal of this paper is to provide explanations for high-level autonomous robot behaviors for which no implementing control program exists. These high-level behaviors include tasks comprising a non-trivial sequence of actions, potentially including reacting to external events and repeated goals; examples include search and rescue missions and the DARPA Urban Challenge [1]. The usual approach to achieving control for such behaviors is to hard-code the high-level aspects, and use path-planning and other low-level techniques during execution. However, with such approaches, it is often not known a priori whether the proposed implementation actually captures the high-level requirements, or whether the intended behavior is even achievable. This motivates the application of formal methods to guarantee that the implemented plans will produce the desired behavior.

A number of frameworks have recently been proposed for the verifiable integration of high-level planning with continuous control. Most rely on an abstraction of the underlying system as a discrete transition system, and use model checking [2] to synthesize control laws (e.g. [3], [4], [5], [6]) on this discrete model. The desired properties are usually expressed using some flavor of temporal logic, such as Linear Temporal Logic (LTL)[7], which is expressive enough to describe specifications that include the desired reactive constraints and sequencing of goals, and allows synthesis of hybrid controllers under several frameworks.

Some recent work has applied efficient synthesis techniques such as [8] to automatically generate provably correct, closed loop, low-level robot controllers that satisfy high-level reactive behaviors specified as LTL formulas [9], [10]. Specifications describe the robot's goals and assumptions on the environment it operates in, using a discrete abstraction. The robot controllers generated represent a rich set of infinite behaviors, and are provably correct-by-construction: the closed loop system they form is guaranteed to satisfy the desired specification in any admissible environment (i.e. any environment that satisfies the modeled assumptions).

In the above formal approaches, when the specification is feasible a controller is generated; however, when there exist admissible environments in which the robot fails to achieve the desired behavior, controller synthesis fails – such a specification is called *unsynthesizable*. An unsynthesizable specification is either *unsatisfiable*, in which case the robot cannot achieve the desired behavior 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 thwart the robot. For example, if the environment can disconnect an otherwise connected workspace, such as by closing a door, a specification requiring the robot to patrol the workspace is merely unrealizable rather than unsatisfiable.

When the specification is unsynthesizable (and there exists no implementing controller), synthesis-based approaches fail to produce the desired behavior, but do not typically provide the user with the exact source of failure. Moreover, even when synthesis is possible, the generated automaton (which fulfills the specification) may produce undesirable or trivial behavior, such as a vacuous controller that does nothing, for reasons involving unsatisfiability or unrealizability of the environment assumptions. This can make troubleshooting a specification an ad hoc and unstructured process. This paper describes an algorithm for automatically analyzing an LTL specification to identify and focus the user's attention on relevant portions thereof. The goal is to enable iterated specification analysis and modification, and facilitate construction of a controller that achieves the user-intended behavior.

By the completeness of the synthesis algorithm in [8], when a specification is unsynthesizable, there exists an admissible environment strategy that demonstrates the system's failure to achieve the specified behavior. This environment strategy is referred to as the counterstrategy, following [11]. In this work, counterstrategy generation is leveraged, along with other tools like Boolean satisfiability testing, to provide explicit feedback on unrealizable specifications in the robot control domain.

Manuscript received January 23, 2012. This work was supported by NSF CAREER CNS-0953365 and ARO MURI (SUBTLE) W911NF-07-1-0216

V. Raman is with the Department of Computer Science, Cornell University, Ithaca, NY 14853 (phone: 607-279-4131; e-mail: vraman@cs.cornell.edu).

H. Kress-Gazit is with the Sibley School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY 14853 (e-mail: hadaskg@cornell.edu).

Feedback is provided by highlighting flawed portions of the user-defined specification (either desired system behavior or environment assumptions), and identifying cases of unexpected and undesirable behavior such as the trivial solutions mentioned earlier. The system and environment components of the specification are considered separately, and checks performed on subcomponents as well as the entire specification. In addition, the specification designer is allowed to interact with the counterstrategy via a domain-specific interface that provides insight on unintuitive reasons for unrealizability by demonstrating the precise environment actions that thwart the robot.

The automated analysis procedures described here are implemented within Linear Temporal Logic MissiOn Planning (LTLMoP)[12], [13], an open source, modular, Python-based toolkit that allows a user to input structured English specifications describing high-level robot behavior, and automatically generates and implements the relevant hybrid controllers using the approach of [9]; the synthesized controllers can be embedded within a simulator or used with physical robots. The most recent version of LTLMoP can be downloaded online[1].

The paper is structured as follows. Section II reviews necessary preliminaries, and Section III provides a formal problem statement. Section IV describes the types of unsatisfiability and unrealizability handled by this work, and provide illustrative examples. Sections V and VI contains the main contributions: an algorithm for identifying unsatisfiable or unrealizable components in an unsynthesizable specification, and an interactive game for exploring the reasons for failure in unrealizable specifications; both sections include examples demonstrating the respective implementations in LTLMoP. Section VII situates this paper in the context of related work. The paper concludes with a description of future directions in Section VIII.

II. Background

The tasks considered in this work involve a robot operating in a known workspace, whose behavior (motion and actions) depends on information gathered at runtime from its sensors about events in the environment. The robot reacts to these events, which are captured by its sensors, in a manner compliant with the task specification, by choosing from a set of actions including moving between adjacent locations. The tasks may also include infinitely repeated behaviors such as patrolling a set of locations.

Example 1. Consider a "hide-and-seek" scenario. The goal is to construct a controller for a robot to play hide and seek in the workspace depicted in Fig 1. The robot starts by counting while the other player hides. When it hears the ready whistle, it takes on the role of seeker and looks for the other player. When it has found the other player, it reverts to counting while the other player hides, and repeats the cycle.

Constructing a controller for this task requires a map of the workspace, in this case a house, with regions of interest marked and labeled. Actions the robot can take are *hiding*, *seeking*, and *counting* while the other player hides. The robot can sense when it has found the target (when in a seeking role), when it has been found (when in a hiding role), and hear the ready whistle when the other player is hiding (i.e. when the robot is in a counting role). Mutual exclusion is required between *hiding*, *seeking*, and *counting*, and between activation of the three sensors (i.e. the robot can never both find the target and be found at the same time). Finally, a formal specification of when the robot takes on the roles of *hiding*, *seeking*, and *counting* is required, along with a description of what these roles entail: when seeking, the robot should visit all rooms until the target has been found; when counting or hiding it can be in any room.

Consider the specification shown in Listing 1, intended to produce a controller for the above behavior. Sentences in a structured language [12] describe the desired robot behavior and assumptions on the environment. However, this specification is unsynthesizable, and there exists no controller to implement the desired behavior in every admissible environment, i.e. every environment that fulfils the specified assumptions; the reason for unsynthesizability is not obvious without further analysis.

Listing 1 Example of unsynthesizable hide-and-seek specification, analyzed in Section IV.

```
# Initial conditions
Env starts with false
Robot starts in porch with counting and not seeking and not
      hiding

# Mutual exclusion of sensors
Always not whistle or not found_target
Always not found_target or not been_found
Always not been_found or not whistle

# Mutual exclusion between roles
Always not seeking or not hiding
Always not hiding or not counting
Always not been_found or not seeking

# Switching between roles
seeking is set on whistle and reset on found_target
hiding is set on found_target and reset on been_found
counting is set on been_found and reset on whistle

# Patrol goals
If you are activating seeking then visit all rooms
If you are not activating seeking then go to any rooms
```

Given the inherent continuous nature of problems in robotics, applying formal methods to the construction of high-level robot controllers requires a discrete abstraction of the problem to enable description with a formal language. Details on the discrete abstraction used in this work can be found in [9]. The formal language used for high-level specifications in this work is Linear Temporal Logic (LTL) [7].

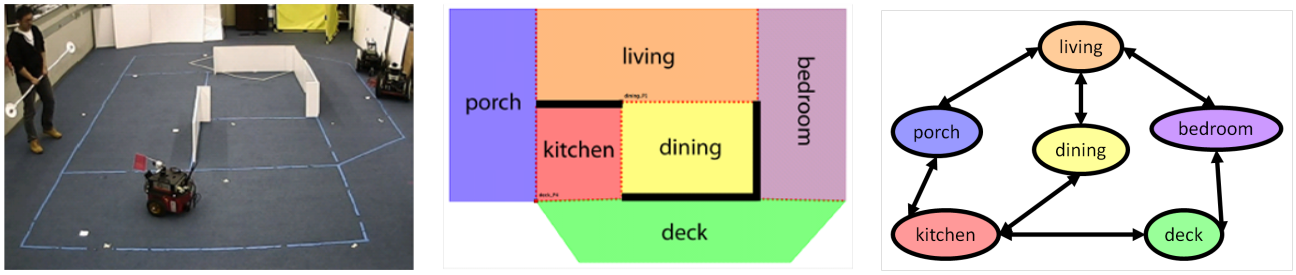
A. Linear Temporal Logic

Syntax: Let AP be a set of atomic propositions. Formulas are constructed from $\pi \in AP$ according to the grammar:

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

where $\neg$ is negation, $\vee$ is disjunction, $\bigcirc$ is "next" , and $\mathcal{U}$ is "until". Boolean constants `True` and `False` are defined as

[1] http://ltlmop.github.com



(a) Real Workspace (b) Polygonal Decomposition, rotated 90°clockwise. Thick lines are walls (c) Adjacency Graph

Fig. 1: Workspace Abstraction and Representation [14]

usual: $\texttt{True} = \pi \vee \neg\pi$ and $\texttt{False} = \neg\texttt{True}$. Conjunction ($\wedge$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), "eventually" ($\Diamond\,\varphi = \texttt{True}\mathcal{U}\varphi$) and "always" ($\Box\,\varphi = \neg\,\Diamond\,\neg\varphi$) are derived.

Semantics: The truth of an LTL formula is evaluated over executions of a finite state machine representing the system. An execution is viewed as an infinite sequence of truth assignments to $\pi \in AP$; a formula is satisfiable if it holds for all executions. Informally, the formula $\bigcirc\,\varphi$ expresses that φ is true in the next "step" or position in the sequence, and the formula $\varphi_1\,\mathcal{U}\,\varphi_2$ expresses the property that φ_1 is true until φ_2 becomes true, and is true only if φ_2 does eventually become true (strong until). The (infinite) truth assignment sequence σ satisfies formula $\Box\,\varphi$ if φ is true in every position of the sequence, and satisfies $\Diamond\,\varphi$ if φ is true at some position of the sequence. Sequence σ satisfies the formula $\Box\,\Diamond\,\varphi$ if φ is true infinitely often. For a formal definition of the semantics, the reader is referred to [2].

LTL is appropriate for specifying robotic behaviors because it provides the ability to describe changes in the truth values of propositions over time. However, to allow users who may be unfamiliar with LTL to define specifications, LTLMoP includes a parser that automatically translates English sentences belonging to a defined grammar [15] into LTL formulas; the grammar includes reactive conditionals, repeated goals, and non-projective locative prepositions such as "between" and "within". This allows users to define desired robot behaviors (including reactive behaviors, e.g., "if you find the target, switch to a hiding role") and specify assumptions about the behavior of the environment (e.g., "the target will never be found in the kitchen") using an intuitive descriptive language rather than the underlying formalism. There are two primary types of properties allowed in a specification – *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 "always" ($\Box$) and "eventually" ($\Diamond$).

B. Discrete Abstraction

Fig. 1 shows the three stages of workspace abstraction for the "hide-and-seek" scenario, from the real environment to a set of convex polygons, and then as a graph with edges connecting adjacent regions. In the discrete abstraction of the problem, the continuous reactive behavior of a robot is described in terms of a finite set of propositions consisting of:

- π_s for every sensor input s (e.g., $\pi_{whistle}$ is true iff the ready whistle is sensed)
- π_a for every robot action a (e.g., $\pi_{counting}$ is true iff the robot is counting)
- π_l for every location l (e.g., $\pi_{bedroom}$ is true iff the robot is in the bedroom).

The set of sensor propositions (controlled by the environment) are denoted by $\mathcal{X}$, and the set of action and location (i.e., robot-controlled) propositions by $\mathcal{Y}$. In Example 1, $\mathcal{X} = \{\pi_{whistle},\ \pi_{found\_target},\ \pi_{been\_found}\}$, $\mathcal{Y} = \{\pi_{porch}, \pi_{deck}, \pi_{bedroom}, \pi_{dining}, \pi_{living}, \pi_{kitchen}, \pi_{hiding}, \pi_{seeking}, \pi_{counting}\}$. Propositions π_{found_target} and π_{been_found} are true when the robot senses that it has found the other player and been found respectively, π_{hiding}, $\pi_{seeking}$, and $\pi_{counting}$ are true depending on the robot's current role in the game. Additionally, the formula $\varphi_l = \pi_l \bigwedge_{l' \neq l} \neg\pi_{l'}$ indicates that the robot is in location l and not in any other location (i.e., locations are mutually exclusive). The possible motion of the robot in the workspace based on the adjacency of the regions is automatically encoded as part of the specification. Legal transitions between adjacent regions are represented as edges between vertices in a graph (with implicit self-loops), and then encoded into a formula over location propositions.

In LTLMoP, in addition to the user-defined specification, topological constraints of the discretized workspace (i.e., which regions are adjacent, and can be moved between) are automatically encoded into the LTL formula to appropriately constrain the possible motions of the robot. This is done by encoding the valid transitions in a formula φ_{trans}; in the above example,

$$
\begin{aligned}
&\varphi_{trans} = \\
&\quad \Box(\varphi_{porch} \Rightarrow \bigcirc(\varphi_{porch} \vee \varphi_{living} \vee \varphi_{kitchen})) \\
&\quad \wedge \Box(\varphi_{deck} \Rightarrow \bigcirc(\varphi_{deck} \vee \varphi_{bedroom} \vee \varphi_{kitchen})) \\
&\quad \wedge \Box(\varphi_{bedroom} \Rightarrow \bigcirc(\varphi_{bedroom} \vee \varphi_{deck} \vee \varphi_{living})) \\
&\quad \wedge \Box(\varphi_{dining} \Rightarrow \bigcirc(\varphi_{dining} \vee \varphi_{living} \vee \varphi_{kitchen})) \\
&\quad \wedge \Box(\varphi_{living} \Rightarrow \bigcirc(\varphi_{living} \vee \varphi_{porch} \vee \varphi_{bedroom} \vee \varphi_{dining})) \\
&\quad \wedge \Box(\varphi_{kitchen} \Rightarrow \bigcirc(\varphi_{kitchen} \vee \varphi_{deck} \vee \varphi_{dining} \vee \varphi_{porch}))
\end{aligned}
$$

C. Controller Synthesis Overview

Fig. 2 provides an overview of the controller synthesis procedure. A user-defined specification and description of the environment topology is automatically parsed into a formula of

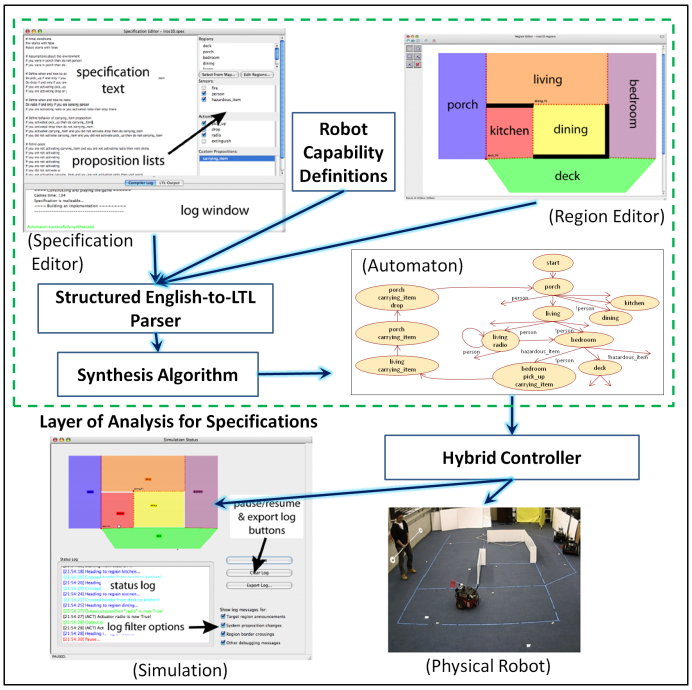


Fig. 2: Controller synthesis overview [13]

the form $\varphi = (\varphi_e \Rightarrow \varphi_s)$, where φ_e encodes any assumption about the sensor propositions, and thus about the behavior of the environment, and φ_s represents the desired behavior of the system. φ_e and φ_s in turn have the structure $\varphi_e = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e$, $\varphi_s = \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s$, where

- φ_i^e and φ_i^s are non-temporal Boolean formulas constraining the initial value(s) for the sensor and system propositions respectively.
- φ_t^e represents assumptions a user may define about possible behaviors of the environment, and consists of a conjunction of formulas of the form $\square A_i$ where each A_i is a Boolean formula with sub-formulas in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc \mathcal{X}$, where $\bigcirc \mathcal{X} = \{\bigcirc x_1, ..., \bigcirc x_n\}$. Intuitively, formula φ_t^e constrains the next sensor values $\bigcirc \mathcal{X}$ based on the current sensor $\mathcal{X}$ and system $\mathcal{Y}$ values. Similarly, φ_t^s represents the robot's required behavior (safety constraints); it consists of a conjunction of formulas of the form $\square A_i$ where each A_i is a Boolean formula in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc \mathcal{X} \cup \bigcirc \mathcal{Y}$ (the system's next state can depend on the environment's current and next states). φ_t^s also contains φ_{trans} as a subformula.
- φ_g^e and φ_g^s represent assumptions on the environment and desired goal behaviors for the system respectively. Both formulas consist of a conjunction of formulas of the form $\square \diamondsuit B_i$ where each B_i is a Boolean formula in $\mathcal{X} \cup \mathcal{Y}$.

In viewing these formulas as corresponding to system and environment properties, this paper refers to φ_t^s and φ_t^e as *safety* properties, and φ_g^s and φ_g^e as *liveness* properties. Listing 2 provides the LTL translation of each sentence of the specification in Listing 1, and identifies the corresponding component of the resulting formula φ.

Given an LTL formula, the synthesis problem consists of constructing an automaton whose behaviors satisfy the formula, if such an automaton exists. For a synthesizable specification φ, synthesis produces an implementing automaton A_φ, enabling the construction of a hybrid controller H_{A_φ} that produces the desirable high-level, autonomous robot behavior. Fig. 3 shows an example of a synthesized automaton for the hide-and-seek problem, using a modified version of the specification in Listing 1, discussed in Section VIII. Each state of the automaton is labeled by the location and action propositions that are true in that state, and each transition is labeled with sensor propositions that must be true for that transition to be enabled. The reader is referred to [8] and [9] for details of the synthesis procedure, and to [9], [12] for a description of how the extracted discrete automaton is transformed into low-level robot control. If no implementing automaton exists, the desired behavior is unsynthesizable.

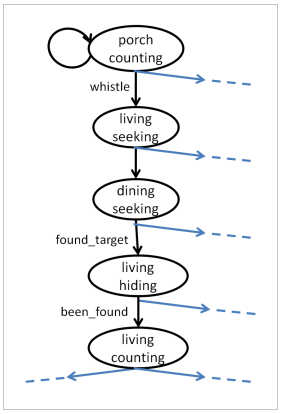


Fig. 3: Excerpt of "hide-and-seek" automaton.

III. Problem Statement

Problem 1. Given a specification $\varphi = (\varphi_e \Rightarrow \varphi_s)$, if there does not exist a non-trivial implementing automaton A_φ, identify the subformulas $\varphi_e^i, \varphi_e^t, \varphi_e^g, \varphi_s^i, \varphi_s^t$ and φ_s^g that are responsible for the unsynthesizability or trivial solution.

Once the problematic subformulas are identified, the corresponding structured English sentences should be highlighted and presented to the user. Additionally, the user should be presented with compelling evidence of the unsynthesizability, enabling them to further understand the cause of failure.

IV. Unsynthesizable Specifications and Undesirable Behavior

The specification in Listing 2 is unsynthesizable, and in particular it is unrealizable, because an adversarial environment can force the robot into a safety violation by setting π_{found_target} to true and $\pi_{whistle}$ to false when $\pi_{counting}$ is true; the robot will turn on its hiding behavior in the next time step ($\bigcirc \pi_{hiding}$ is set by line 13), but still be counting ($\bigcirc \pi_{counting}$ required by line 12), and therefore simultaneously satisfy $\bigcirc \pi_{hiding} \wedge \bigcirc \pi_{counting}$, violating the safety condition in line 7 – the system thus has no legal next state.

Listing 2 Unsynthesizable specification from Listing 1, with corresponding LTL translation

	# Environment initial condition	Component of φ_i^e
1	Env starts with **false**	$\neg\pi_{whistle} \wedge \neg\pi_{found\_target} \wedge \neg\pi_{been\_found}$
	# Robot initial condition	Component of φ_i^s
2	Robot starts in **porch** with **counting** and not **seeking** and not **hiding**	$\varphi_{porch} \wedge \pi_{counting} \wedge \neg\pi_{seeking} \wedge \neg\pi_{hiding}$
	# Assumptions about the environment – mutual exclusion of sensors	Component of φ_t^e
3	Always not **whistle** or not **found_target**	$\square(\neg \bigcirc \pi_{whistle} \vee \neg \bigcirc \pi_{found\_target})$
4	Always not **found_target** or not **been_ found**	$\square(\neg \bigcirc \pi_{found\_target} \vee \neg \bigcirc \pi_{been\_found})$
5	Always not **been_ found** or not **whistle**	$\square(\neg \bigcirc \pi_{been\_found} \vee \neg \bigcirc \pi_{whistle})$
	# Robot safety – mutual exclusion between roles	Component of φ_t^s
6	Always not **seeking** or not **hiding**	$\square(\neg \bigcirc \pi_{seeking} \vee \neg \bigcirc \pi_{hiding})$
7	Always not **hiding** or not **counting**	$\square(\neg \bigcirc \pi_{hiding} \vee \neg \bigcirc \pi_{counting})$
8	Always not **been_ found** or not **seeking**	$\square(\neg \bigcirc \pi_{counting} \vee \neg \bigcirc \pi_{seeking})$
	# Robot safety – switching between roles	Component of φ_t^s
9	**seeking** is set on **whistle** and reset on **found_ target**	$\square(\pi_{whistle} \rightarrow \bigcirc \pi_{seeking})$ $\square(\pi_{found\_target} \rightarrow \bigcirc \neg\pi_{seeking})$ $\square(\pi_{seeking} \wedge \neg\pi_{found\_target} \rightarrow \bigcirc \pi_{seeking})$ $\square(\neg\pi_{seeking} \wedge \neg\pi_{whistle} \rightarrow \bigcirc \neg\pi_{seeking})$
10	**hiding** is set on **found_ target** and reset on **been_ found**	$\cdots$
11	**counting** is set on **been_ found** and reset on **whistle**	$\cdots$
	# Patrol goals	Component of φ_g^s
12	If you are activating **seeking** then visit **all rooms**	$\bigwedge_{r\in\text{locations}} \square \diamondsuit(\pi_{seeking} \rightarrow \varphi_r)$
13	If you are not activating **seeking** then go to **any rooms**	$\square \diamondsuit (\neg\pi_{seeking} \rightarrow \bigvee_{r\in\text{locations}} \varphi_r)$

A. Unsynthesizable Categories

As mentioned before, there are several possibilities to be considered when reasoning about a specification that cannot be synthesized, or one that results in generation of a controller that does not behave as intended.

1) Unsatisfiability: Consider this simple illustrative specification in the hide-and-seek scenario:

Always not **porch**	$\square \neg \bigcirc \varphi_{porch}$	(in φ_t^s)
Visit **porch**	$\square \diamondsuit \varphi_{porch}$	(in φ_g^s)

This specification is not synthesizable, and in particular it is unsatisfiable, since φ_t^s and φ_g^s are inconsistent no matter what the environment does, and so the system has no winning strategy.

2) Unrealizability: Now consider the following specification (separate from the one above):

If you are sensing **whistle** then do **porch**		
	$\square(\pi_{whistle} \Rightarrow \bigcirc \varphi_{porch})$	(in φ_t^s)

Depending on the current location, φ_{trans} does not always allow $\bigcirc \varphi_{porch}$. For example, if the robot hears the whistle in bedroom, it cannot reach porch in the next discrete step (since it has to pass another region first). Therefore, if the system hears the whistle, then there may be no further transitions that satisfy the robot safety; the environment can therefore win from some initial states (e.g. bedroom) by making $\pi_{whistle}$ true. This specification is *unrealizable*, but not *unsatisfiable* – there are environment strategies for which the system achieves the desired behavior, such as the environment that never sets $\pi_{whistle}$ to true.

Symmetric to system unrealizability is the case where a winning system strategy prevents the environment from satisfying the formula φ_e. Overloading terminology, the environment is termed unrealizable in this case. For example, if the environment safety condition in the above example were to include "If you were in porch then do not person and do person", then the environment would be unrealizable if the system can go to the porch, and the system would win regardless of whether it fulfilled its goals.

3) Undesirable Behavior after Synthesis: Consider the same map again, with the following specification:

Always not **porch**	$\square \neg \bigcirc \varphi_{porch}$	(in φ_t^s)
Visit **porch**	$\square \diamondsuit \varphi_{porch}$	(in φ_g^s)
Sense **whistle** and do not sense **whistle**		
	$\square(\bigcirc \pi_{whistle} \wedge \bigcirc \neg\pi_{whistle})$	(in φ_t^e)

Here φ_t^e is unsatisfiable, so φ_e is unsatisfiable. Since the antecedent of the implication is always false, the formula $\varphi_e \implies \varphi_s$ is satisfied by any automaton, and all initial states are winning for the system, even though the system itself is also unsatisfiable. The algorithm in [8] returns a trivial automaton consisting of all the initial states, but no transitions between states; in the case of the hide-and-seek example, this is a single state, in which the robot is counting in the porch. Since each state in the automaton has an implicit self-loop in the continuous level implementation, a controller based on this automaton would cause the robot to stay in the porch

indefinitely – this is likely not the user-intended behavior.

The above examples demonstrate that when the algorithm does not return an automaton, there is some ambiguity in what went wrong. In particular, an unsynthesizable specification $\varphi_e \implies \varphi_s$ is either *unrealizable* or *unsatisfiable*. Even once these two cases are distinguished, there is further ambiguity surrounding the cause of the unsatisfiability/unrealizability of the specification. In addition, if φ_e is unsatisfiable, then a trivial automaton is obtained.

B. Causes of Failure

Section IV-A detailed the distinction between unsatisfiable and unrealizable specifications. In either case, failure occurs either if it is possible for the environment to steer the corresponding two-player game into a state from which the system has no valid move (as in Example IV-A2), or if the environment can prevent the system from satisfying one of the liveness conditions (goals) (as in Example IV-A1). The former case is termed *deadlock*, and the latter case *livelock*.

C. Identifying Deadlock

Identifying deadlock calls for a characterization of the set of "bad" states from which the environment can force the system into a state such that every transition will violate the system safety (in which case the system has no next move in the game in [8]). Such a characterization of states can be expressed in the modal μ-calculus, which extends propositional modal logic with least and greatest fixpoint operators μ, ν [16]. The μ-calculus over game structures is defined as in [8]. A formula φ is interpreted as the set of states $[\![\varphi]\!]$ in which φ is true. Under this interpretation, the logical operator $\textcircled{\diamond}$ is defined such that a state s is included in $[\![\textcircled{\diamond}\,\varphi]\!]$ if the system can force the play to reach a state in $[\![\varphi]\!]$, regardless of how the environment moves from s. For example, if the environment safety condition includes "If you were in porch then do not whistle", then any state in which the system is in the porch would be included in $[\![\textcircled{\diamond}\,\neg\pi_{whistle}]\!]$, since the environment cannot activate $\pi_{whistle}$ in the next state. Similarly, operator $\textcircled{\Box}$ is defined such that a state s is included in $[\![\textcircled{\Box}\,\varphi]\!]$ if the environment can force the play to reach a state in $[\![\varphi]\!]$.

The set of "bad" states is now characterized by the fixpoint formula $\mu X.\ X \vee \textcircled{\Box}\, X$, and constructed by having X initialized to **FALSE** and updated at each iteration with $X \leftarrow X \vee \textcircled{\Box}\, X$ until two iterations are identical. Intuitively, at each iteration of the fixpoint computation, the construction adds in states such that the environment can force the system into the "bad" set. The fixpoint set therefore characterizes all states that can reach a system safety violation. If this set of bad states intersects the initial states, then there is some initial state from which the environment can eventually force the system to violate its safety conditions, thereby winning the game. Similarly, $\mu X.\ X \vee \textcircled{\diamond}\, X$ characterizes the set of states from which the system can force the environment into deadlock.

V. Algorithm for Analysis of Specifications

This section describes in detail the steps of of Algorithm 1 introduced in [14], for isolating sources of unsatisfiability/unrealizability in the system and environment components of an unsynthesizable or trivial specification; this paper provides details that were omitted from previous work for brevity. Given a specification, properties of the synthesis problem are leveraged to determine whether each of φ_e and φ_s is unrealizable or unsatisfiable, and present the user with this information. Given the input specification parsed into a suitable representation of the environment (φ_e) and robot (φ_s) LTL formulas, a series of tests are applied to determine whether each is unrealizable or unsatisfiable. In LTLMoP, the presented algorithm is implemented in the JTLV framework [17], with the corresponding formulas for the initial conditions, transitions and goals represented as Binary Decision Diagrams (BDDs) [18]. The BDD corresponding to a formula is a compact representation of the set of proposition value combinations that satisfy that formula; this compact representation enables efficient operations on these sets.

A. Synthesis and trivial automata

The following pseudocode describes the initialization of variables from Algorithm 1 in [14]. If the specification φ is realizable, a BDD representation of the set of all implementing control automata AUT_SET is synthesized using the $SYNTHESIS$ algorithm from [8], and a single such automaton AUT is extracted. Note that in the BDD representation, **FALSE** denotes the empty set, and **TRUE** denotes the set of *all* automata. Otherwise, a set of all possible counterstrategies CTR_SET is obtained following a construction $COUNTERSTRATEGY$, adapted from that presented in [11]. If an automaton is synthesized, but has no transitions (i.e. is trivial), the user is alerted to this fact.

Algorithm 1 Initialization of variables from Algorithm 1 in [14]

$\varphi_t^p = \bigwedge_j \Box A_j^p, \varphi_g^p = \bigwedge_{i=1}^{n_g^p} \Box \Diamond B_i^p$ for $p \in \{s, e\}$
$AUT\_SET \leftarrow SYNTHESIS(s, e)$
if AUT_SET != **FALSE** (spec. is synthesizable) **then**
    $AUT \leftarrow AUT\_SET$
    if AUT has no transitions **then**
        flag as *trivial*
else
    $CTR\_SET \leftarrow COUNTERSTRATEGY(s, e)$

B. Unsatisfiable initial conditions and transition relations

Recall that φ_i^e and φ_i^s consist of a conjunction of formulas of the form $\Box\, A_i$ where each A_i is a Boolean formula, so for either of these, an emptiness check on the BDD representing the set of variable assignments satisfying $\varphi_i^p \bigwedge_i A_i$ determines whether the transitions in a single time step are satisfiable from the initial condition. The following pseudocode checks for the unsatisfiability of the initial conditions and the transitions relation (safety) for both environment and system.

1) Unsatisfiability of transition relations over multiple steps: The above check will not identify unsatisfiability of following the transitions over multiple time steps; for example, the transition relation $\Box(\pi_{hiding} \implies \bigcirc \pi_{hiding}) \wedge \Box(\pi_{hiding} \implies \neg \bigcirc \pi_{hiding}) \wedge \Box(\neg\pi_{hiding} \implies$

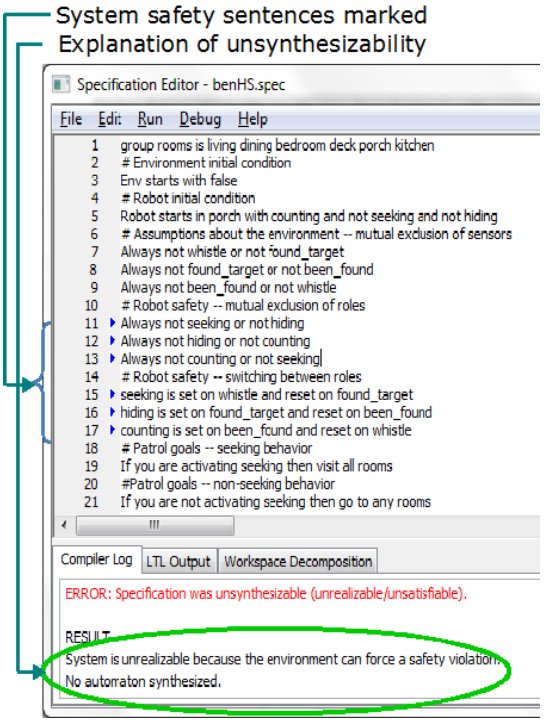


Fig. 4: Analyzing an unsynthesizable specification

Algorithm 2 Initial conditions and transition unsatisfiability tests from Algorithm 1 in [14]

9: **if** φ_i^p == **FALSE then**
10: player p has *unsatisfiable initial conditions*
11: **if** $\varphi_i^p \wedge \bigwedge_i A_i^p$ == **FALSE then**
12: p has *unsatisfiable transitions*

$\bigcirc \pi_{hiding}$) is unsatisfiable when starting from the initial condition $\neg\pi_{hiding}$, because π_{hiding} is true in the second time step leading to no valid transitions (since any valid transition would have to satisfy both π_{hiding} and $\neg\pi_{hiding}$); however the analysis so far will not detect this. Such "multi-step" unsatisfiability of the transitions is identified by computing the set of environment counterstrategies (i.e. the strategies the environment can use to find sensor inputs such that there is no robot response fulfilling the specification), using the counterstrategy synthesis algorithm in [11]. If every sequence of environment moves is in this counterstrategy, then the system must be unsatisfiable. In addition, if every sequence of environment moves forces the system into deadlock (rather than livelock), the system safety is unsatisfiable; this is identified using a fixpoint computation as described earlier. The symmetric case for multi-step unsatisfiable environment transitions looks at the set of system winning strategies and checks that every sequence of system actions is winning.

Algorithm 3 Multi-step unsatisfiability tests from Algorithm 1 in [14]

13: **if** $\forall\sigma \in CTR\_SET$ (resp. $\forall\sigma \in AUT\_SET$), σ leads to deadlock **then**
14: s (resp. e) *transitions are unsatisfiable from initial conditions*

C. Unsatisfiable goals

The next steps of the algorithm check for unsatisfiability of system and environment liveness conditions. Any liveness

Algorithm 4 Unsatisfiable goal tests from Algorithm 1 in [14]

15: **for** $i := 1$ to n_g^p **do**
16: **if** B_i^p == **FALSE then**
17: p *goal* i *is unsatisfiable*
18: **for** $i := 1$ to n_g^p **do**
19: **if** $B_i^p \wedge \bigwedge_j A_j^p$ == **FALSE OR** $\bigcirc B_i^p \wedge \bigwedge_j A_j^p$ == **FALSE then**
20: p *is unsatisfiable between goal* i *and transitions*

condition φ_g^p consists of a conjunction of clauses of the form $\square \diamondsuit B_i$, and the safety φ_t^p consists of a conjunction of formulas of the form $\square A_j$. So a contradiction in $\varphi_g^p \wedge \varphi_t^p$ implies that some $\square \diamondsuit B_i$ is inconsistent with $\bigwedge_j \square A_j$, i.e., $\diamondsuit B_i$ is inconsistent with $\bigwedge_j \square A_j$. The A_js in the transition formula govern proposition values in the current and next time steps (as described in Section II-B), and in order for a goal B_i to be satisfied infinitely often, it needs to be consistent with each A_j in both the current and next time steps (so there are valid transitions into and out of each goal state). This is confirmed by checking $B_i \wedge \bigwedge_j A_j$ and $\bigcirc B_i \wedge \bigwedge_j A_j$ for consistency – if either is inconsistent, then liveness condition B_i cannot be fulfilled infinitely often while following the transitions allowed by the safety.

1) Unsatisfiability of goals over multiple steps: The test in Algorithm 4 is once again not complete, and detecting multi-step unsatisfiability of the system (resp., the environment) requires checking that every counterstrategy (resp., every robot strategy) leads to livelock for the system (resp., the environment). If the system is unsatisfiable due to livelock, the environment can "lock" the system out of some liveness; the faulty liveness can be identified by starting with no liveness conditions and including them incrementally until synthesis fails (this involves running the synthesizability check once for each liveness condition, as in lines 23-31 in 5).

Algorithm 5 Unsatisfiable and unrealizable goal tests from Algorithm 1 in [14]

23: **for** $i := 1$ to n_g^s **do**
24: $\varphi_g^{s_i} = \bigwedge_{k=1}^{i} \square \diamondsuit B_k^s, \varphi_t^{s_i} = \varphi_t^s, \varphi_i^{s_i} = \varphi_i^s$
25: $AUT\_SET_i \leftarrow SYNTHESIS(s_i, e)$
26: **if** AUT_SET_i == **FALSE** (unsynthesizable) **then**
27: $CTR\_SET_i \leftarrow COUNTERSTRATEGY(s_i, e)$
28: **if** CTR_SET == **TRUE then**
29: i^{th} *system goal inconsistent with transition relation*
30: **else if** AUT_SET_{i-1} != **FALSE then**
31: i^{th} *system goal is unrealizable*
32: **for** $i := n_g^e$ to 1 **do**
33: $\varphi_g^{e_i} = \bigwedge_{k=i}^{n_g^e} \square \diamondsuit B_k^e, \varphi_t^{e_i} = \varphi_t^e, \varphi_i^{e_i} = \varphi_i^e$
34: $AUT\_SET_i \leftarrow SYNTHESIS(s, e_i)$
35: **if** AUT_SET_i != **FALSE** (synthesizable) **then**
36: **if** AUT_SET == **TRUE then**
37: i^{th} *environment liveness inconsistent with transitions*
38: **else if** AUT_SET_{i+1} == **FALSE then**
39: i^{th} *environment liveness condition is unrealizable*

If the environment counterstrategy is **TRUE**, then the sys-

tem is in fact unsatisfiable due to the most recently added goal (lines 28-29). A symmetric test for the environment runs the synthesis algorithm starting with all environment liveness conditions, and removes them one by one (lines 32-39). Similarly, if every system strategy is winning, the current environment goals must be unsatisfiable, and if removing an environment liveness condition makes the specification unsynthesizable, the system's winning strategy involved falsifying the removed environment liveness (36-37).

If the algorithm does not detect system unsatisfiability, the system might still be unrealizable. To win from an initial state of the game, the environment must either force the system into deadlock (i.e. a safety violation) or livelock, in which case the unrealizability is because some goal(s) cannot be achieved while adhering to the transitions specified. As described earlier, reachability analysis using fixpoint operators suffices to check that there is some sequence of environment actions that forces the system into deadlock, and likewise for the system forcing the environment into deadlock, as follows:

Algorithm 6 Deadlock tests from Algorithm 1 in [14]

20: **if** $\exists\sigma \in CTR\_SET$ (resp. $\exists\sigma \in AUT\_SET$), σ leads to deadlock) **then**
21: s (resp. e) is *unrealizable as it can be forced into a safety violation*

If the (unrealizable) system cannot be forced into a safety violation, there exists and environment strategy to "lock" the system out of some liveness; the faulty liveness can be identified as in the unsatisfiable case, requiring only *some* (and not every) counterstrategy to be winning, as in lines 29-30; the symmetric test for environment livelock is in lines 37-38.

Example 2. Consider again the hide-and-seek specification in Listing 1. As mentioned in Section III, this specification fails to produce a controller, and it is difficult to determine where the problem is without the aid of the presented analysis. The proposed tests determine that the system is unrealizable because the environment can force a safety violation. This allows the user to focus their attention on the relevant sentences (highlighted as in Fig. 4).

D. Guarantees:

The algorithm presented above is complete for system unsynthesizability, in the sense that every incidence of system unsatisfiability or unrealizability falls into one of the handled cases. Note that the algorithm provides information about both system and environment components. By notifying the user if the environment is unsatisfiable or unrealizable, they are alerted to the fact that the behavior generated may not be as intended, prior to execution. However, there are cases of environment unsatisfiability or unrealizability that may not be identified by the above tests. When the environment is unrealizable because of livelock, but the system itself is deadlocked, the system has no infinite strategies, and therefore cannot cause environment livelock. Additionally, if the system is realizable independent of the environment, the tests in Algorithm 5 will not reveal any information about the environment, since synthesis will never fail. However in this case, following the system strategy construction in [8], the system will achieve the desired behavior rather than prevent the environment from fulfilling its goals, so the environment unrealizability has no consequences for the robot's behavior. All other cases of environment unrealizability are captured by the above tests.

VI. Interactive Exploration of Unrealizable Specifications

The algorithm described in the previous section enables highlighting of sentences of the specification that contribute to the unsynthesizability. However, so far the user is not presented with any evidence to show that the specification cannot be implemented. In the case of deadlock, it may be possible to present the user with a set of finite move sequences leading to a safety violation. However, in the case of livelock, possibly exhaustive set of move sequences sequences of moves are needed to demonstrate livelock. This problem is addressed in this work via an interactive game. The counterstrategy synthesis algorithm introduced in [11] is used to extract a strategy for the environment, and find sequences of environment actions such that there is no robot response fulfilling the specification. The user interacts with this strategy by selecting the robot actions and movement in every time step in response to the sensor inputs provided by LTLMoP. The user can change the state of robot actuators by clicking toggle buttons, and select a region to move to by clicking on a map. The available choices are automatically constrained according to the system safety conditions: forbidden regions are blacked out on the map, and illegal action choices raise an error, as shown in Fig. 5(d).

Example 3. In the case of the unsynthesizable hide-and-seek example from Listing 1, the cause of unrealizability is evident at the very first state, as shown in Fig. 6. The environment has set π_{found_target} to true, and there are no safe robot actions from the displayed state, as indicated by the error message on the screen.

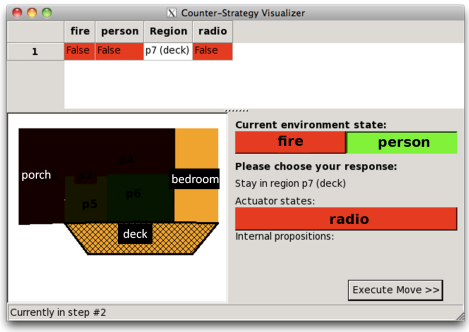
Example 4. Consider the specification in Listing 3, drawn from the "fire-fighting" scenario introduced in [14]. The robot task is to enter the house depicted in Fig. 1 from the deck and visit the porch infinitely often. If it encounters a person, it cannot move directly to the kitchen. Similarly, if it senses fire, it cannot move to the living room. The radio is always turned off, and the assumption on the environment is that a person will never be sensed simultaneously with fire.

Listing 3 Example of unrealizable specification for counterstrategy visualization.

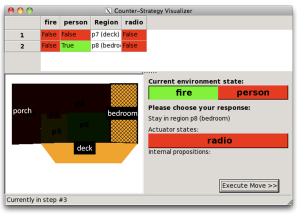
```
1 Robot starts with false
2 Robot starts in deck
3 Visit porch
4 If you are sensing person then do not kitchen
5 If you are sensing fire then do not living
6 Always do not (fire and person)
7 Always do not radio
```

The environment can prevent the robot from satisfying its goal (to visit the porch infinitely often) by alternately enabling π_{fire} and π_{person}, thereby trapping the robot in the deck and bedroom, i.e. away from the porch. The first three steps of this (infinite) counterstrategy are shown being played through in the Counterstrategy Visualization Tool in Figure 5. In the tool, regions that cannot be chosen due to the motion constraints are black.

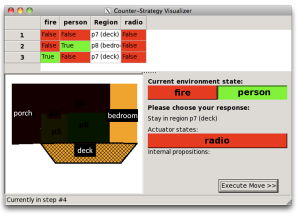
The first step is the setting of a valid initial condition from which the environment can win: the robot is in the deck and all other sensor and action propositions are set to false. Second, the environment enables π_{person} so the robot cannot enter the kitchen and is in the next step confined (by the adjacency graph in Fig. 1(c)) to the deck and bedroom as depicted in Fig. 5(a). The user responds by moving the robot to the bedroom, and so the environment then switches to enabling π_{fire} and disabling π_{person} as required by line 6 (Fig. 5(b)); this prevents the robot from entering the living room, and the user returns to the deck. This move results in the original configuration, and the environment again switches on π_{person} and turns off π_{fire}, as shown in Fig. 5(c); at this point it should be clear to the user why the task is unrealizable – the environment can keep it out of the porch.



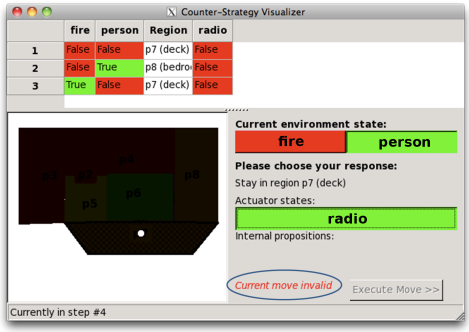
(a) Robot starts in the $deck$, environment's first move turns $person$ on.



(b) User moves the robot to the $bedroom$; environment turns off $person$ and turns on $fire$, preventing $living$.



(c) User moves the robot back to the $deck$, environment turns off $fire$ and turns on $person$, and is back where it started.



(d) An error message is displayed if the user tries to select $radio$ for the robot.

Fig. 5: Interactive Game for Unrealizable Specifications [14]

VII. Related Work

There are several frameworks that use temporal logics for robot control, including those based on model checking [3], [4], [5], [6] and synthesis [9], [10]. However, the problem of explaining robot behaviors that cannot be achieved has only recently been addressed [13], [14], [19]. This paper supersedes the work described in [13], [14], and includes additional original examples illustrating the full specification analysis procedure. In addition, formal guarantees are discussed with respect to the algorithm presented. The work in [19] addresses the problem of revising LTL specifications that are not satisfied on a given system. The author defines a partial order on LTL formulas, and defines the notion of a valid relaxation for an LTL specification, which informally corresponds to the set of formulas larger than all the formulas in that specification. They then present formula relaxation for unreachable states, which is accomplished by recursively removing all positive occurrences of unreachable propositions in a manner similar to the fixpoint calculation described in Section V. On the other hand, to revise specifications with logical inconsistencies, they take the synchronous product of the system and environment specifications and add in disallowed transitions as needed to achieve the goal state. The work presented in this paper differs in its objective, which is to provide feedback on existing specifications, not rewrite them. The main advantage of the approach in this paper over that in [19] is that the inconsistencies are mapped directly to sentences in the specification language. Moreover, the techniques applied in this paper deal with reactive specifications.

While explaining unachievable robot behaviors is a relatively new area of research in robotics, there has been prior work on unsatisfiability and unrealizability of LTL in the formal methods literature. Some approaches focus on analyzing

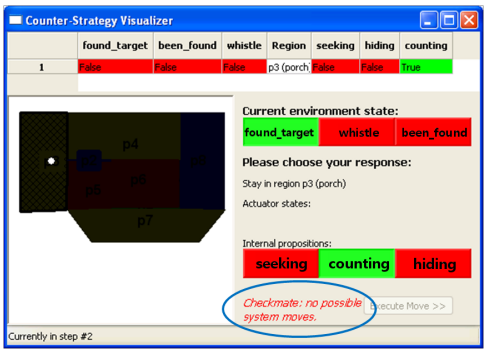


Fig. 6: Counterstrategy visualization for "hide-and-seek" example. The circled message reads, "Checkmate: no possible system moves".

unsatisfiable LTL formulas without considering unrealizability, and exploiting existing tools like model checkers [20]. Others investigate notions of unsatisfiable core for an LTL formula[21], and use techniques like Boolean enumeration and temporal reasoning to search for these cores [22]. For model-checking based unsatisfiability testing, [23] use formal definitions of causality to explaining counterexamples provided by model-checkers; they detect a set of causes of failure conjectured to be the union of the minimal unsatisfiable cores, but more appropriate to understanding a counterexample.

Similarly, there has been work on checking that a satisfiable LTL specification can be implemented (i.e., is realizable). On the diagnostic front, the authors of [24] propose definitions for "helpful" assumptions and guarantees, and compute minimal explanations of unrealizability (i.e., "unrealizable cores") by iteratively expelling unhelpful constraints. The corresponding problem of correcting a general unrealizable LTL specification has been approached in [25], by computing the minimal additional environment assumptions that would make an unrealizable specification realizable, and implemented using efficient analysis of turn-based probabilistic games.

The research presented in this paper is the first work on analysing high-level specifications in the robotics domain. The techniques applied are most closely related to [11], whose authors implement a set of sophisticated specification analyses in an interactive tool, RATSY [26], which demonstrates unrealizability in hardware design specifications. The interactive game presented in this paper is better adapted to the robot domain, as described in Section VI.

VIII. Conclusions and Future Work

This paper addresses the problem of explaining the cause of failure in high-level autonomous robot specifications for which there either does not exist an implementing controller, or the implementation is trivial. An algorithm is presented for systematically analyzing robot behavior specifications, exploiting the structure of the specification to narrow down possible reasons for failure to create a robot controller. The approach is implemented as part of the open source LTLMoP toolkit. The synthesis process is enclosed in a layer of reasoning that identifies the cause of failure, enabling the user to target their attention to the relevant portions of the specification. The user is also allowed to explore the cause of failure in an unsynthesizable specification by means of an interactive game.

Future work will leverage existing techniques to further isolate the source of failure and provide the user with comprehensive feedback, including modifications to the input that would allow synthesis. If the presented algorithm determines that the system or environment is unsatisfiable, further analysis can narrow down the cause of this unsatisfiability by finding an unsatisfiable core. Similarly, if the system is unrealizable, analysis can proceed by computing the unrealizable core, or adding additional environmental assumptions [25], [19].

For instance, the specification in Example 3 can be made realizable by adding assumptions on the environment to prevent it from setting π_{found_target} to true and $\pi_{whistle}$ to false when $\pi_{counting}$ is true. Additional assumptions, shown in Listing 4, are also needed to prevent the environment from causing safety violations when π_{hiding} and $\pi_{seeking}$ are true. A key direction of future research is the development of efficient techniques for computing unsatisfiable and unrealizable cores and added assumptions for specifications in the robotics domain.

Listing 4 Additional environment assumptions that make the specification in Listing 1 synthesizable

```
If you were activating counting then do not
    found_target and not been_found
If you were activating seeking then not been_found
    and do not whistle
If you were activating hiding then do not whistle
    and not found_target
```

References

[1] M. Buehler, K. Iagnemma, and S. Singh, Eds., *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic, George Air Force Base, Victorville, California, USA*, ser. Springer Tracts in Advanced Robotics, vol. 56. Springer, 2009.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[3] M.Kloetzer and C.Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Transaction on Automatic Control*, vol. 53, no. 1, pp. 287–297, 2008.

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

[5] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals," in *IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 2689–2696.

[6] L. Bobadilla, O. Sanchez, J. Czarnowski, K. Gossman, and S. LaValle, "Controlling wild bodies using linear temporal logic," in *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[7] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE, 1977, pp. 46–57.

[8] N. Piterman and A. Pnueli, "Synthesis of reactive(1) designs," in *In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI 06*. Springer, 2006, pp. 364–380.

[9] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[10] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon control for temporal logic specifications," in *Hybrid Systems*, 2010, pp. 101–110.

[11] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications using simple counterstrategies," in *Formal Methods in Computer-Aided Design*, 2009, pp. 152–159.

[12] C. Finucane, G. Jing, and H. Kress-Gazit, "LTLMoP: Experimenting with language, temporal logic and robot control," in *IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, 2010, pp. 1988 – 1993.

[13] V. Raman and H. Kress-Gazit, "Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP," in *CAV*, 2011, pp. 663–668.

[14] V. Raman and H. Kress-Gazit, "Automated feedback for unachievable high-level robot behaviors," in *ICRA*, 2012 (To appear).

[15] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Translating structured english to robot controllers," *Advanced Robotics*, vol. 22, no. 12, pp. 1343–1359, 2008.

[16] D. Kozen, "Results on the propositional mu-calculus," *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[17] A. Pnueli, Y. Sa'ar, and L. D. Zuck, "JTLV: A framework for developing verification algorithms," in *Computer Aided Verification*, 2010, pp. 171–174.

[18] C. Lee, "Representation of switching circuits by binary-decision programs," *Bell Systems Technical Journal*, vol. 38, p. 85999, 1959.

[19] G. E. Fainekos, "Revising temporal logic specifications for motion planning," in *ICRA*. IEEE, 2011, pp. 40–45.

[20] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," in *14th Workshop on Model Checking Software (SPIN '07)*, ser. Lecture Notes in Computer Science (LNCS), vol. 4595. Springer-Verlag, 2007, pp. 149–167.

[21] V. Schuppan, "Towards a notion of unsatisfiable cores for LTL," in *Fundamentals of Software Engineering, Third IPM International Conference*, 2009, pp. 129–145.

[22] A. Cimatti, M. Roveri, V. Schuppan, and S. Tonetta, "Boolean abstraction for temporal logic satisfiability," in *CAV*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 532–546.

[23] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler, "Explaining counterexamples using causality," in *Computer Aided Verification*, 2009, pp. 94–108.

[24] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev, "Diagnostic information for realizability," in *VMCAI*, ser. Lecture Notes in Computer Science, F. Logozzo, D. Peled, and L. D. Zuck, Eds., vol. 4905. Springer, 2008, pp. 52–67.

[25] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *Proceedings of the 19th international conference on Concurrency Theory*, ser. CONCUR '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 147–161. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85361-9_14

[26] R. P. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber, "RATSY - a new requirements analysis tool with synthesis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, Springer, Ed., vol. 6174, 2010, pp. 425 – 429.

Vasumathi Raman is a Ph.D. candidate in Computer Science at Cornell University. Her research focuses on explaining unachievable robot behaviors in the context of automated synthesis of controllers for high-level tasks. She previously obtained a B.A. from Wellesley College in 2007, with majors in Computer Science and Mathematics.

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 was a finalist for the Best Student Paper Award at ICRA 2007 and a finalist for the Best Paper award at IROS 2007.