Automated Feedback For Unachievable High-Level Robot Behaviors

Vasumathi Raman
Department of Computer Science
Cornell University
Ithaca, NY 14853, USA
(vraman@cs.cornell.edu)

Hadas Kress-Gazit
Sibley School of Mechanical and
Aerospace Engineering
Cornell University
Ithaca, NY 14853, USA
(hadaskg@cornell.edu)

***Abstract*— One of the main challenges in robotics is the generation of controllers for autonomous, high-level robot behaviors comprising a non-trivial sequence of actions. 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 there is no controller that fulfills the specification, these approaches do not provide the user with a source of failure, making the troubleshooting of specifications an unstructured and time-consuming process. In this paper, we describe a procedure for analyzing an unsynthesizable specification to identify causes of failure. We also provide an interactive game for exploring possible causes of failure, in which the user attempts to fulfill the robot specification against an adversarial environment. Our approach is implemented within the LTLMoP toolkit for robot mission planning.**

I. Introduction

Our goal is to provide feedback on high-level autonomous robot behaviors for which no implementing controller exists. Such high-level behaviors comprise non-trivial sequences of actions, including reacting to external events and repeated goals. Examples include search and rescue missions and the DARPA Urban Challenge [4]. A number of frameworks have recently been proposed for the verifiable integration of high-level planning with continuous control, to guarantee that the implemented plans will produce the desired behavior. Most rely on an abstraction of the underlying system as a discrete transition system, and use model checking [8] to synthesize control laws (e.g. [16], [2]) on this discrete model. The desired properties are usually expressed using some flavor of temporal logic, such as Linear Temporal Logic (LTL)[18].

Some recent work [14], [23] has applied efficient synthesis techniques [17] to automatically generate provably correct, closed loop, low-level robot controllers that satisfy high-level reactive behaviors specified as LTL formulas. Specifications describe the robot's goals and assumptions on the environment it operates in, using a discrete abstraction. The hybrid 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. one that satisfies the modeled assumptions).

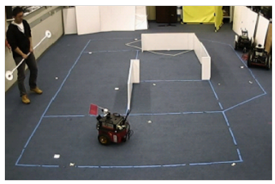
In the above formal approaches, when a specification is feasible, a controller is generated; however, when there exist admissible environments in which the robot fails to achieve the desired behavior, controller synthesis fails – we call such a specification *unsynthesizable*. An unsynthesizable specification is *unsatisfiable* if the robot cannot achieve the desired behavior no matter what happens in the environment, and *unrealizable* if there exists at least one environment that can thwart the robot. An example of unsatisfiability is a disconnected workspace, where the robot goal is to patrol all regions; on the other hand, if there is a door the environment could close to disconnect an otherwise connected workspace, the specification is unrealizable.

When the specification is unsynthesizable, 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 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 a procedure for systematically analyzing an LTL specification to identify and focus the user's attention on relevant portions thereof.

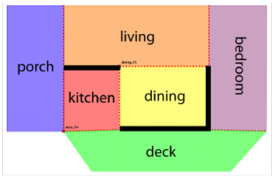
II. Related Work

There has been previous work on analyzing unsatisfiable LTL formulas in the formal methods literature [21], [7], [22], [1]. Similarly, there has been work on explaining why satisfiable specifications cannot be implemented because of properties of the environment (i.e., unrealizable specifications) [6], [5]. Our approach is most closely related to [11] and the derived interactive tool RATSY [3], which demonstrates unrealizability in hardware design specifications. We use a similar counterstrategy generation approach to explain unrealizable specifications in the robot control domain, by allowing the specification designer to interact with a simulated environment that demonstrates the problem. In addition, we provide explicit feedback on specification components, 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 in addition to the entire specification.

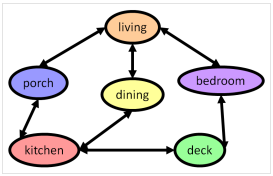
This work was supported by NSF CAREER CNS-0953365 and ARO MURI (SUBTLE) W911NF-07-1-0216



(a) Real Workspace



(b) Polygonal Decomposition, rotated 90° clockwise. Thick lines are walls



(c) Adjacency Graph

Fig. 1: Workspace Abstraction and Representation

In the robotics domain, the problem of revising LTL specifications that are not satisfied on a given system was recently addressed in [9]. In contrast, the work presented in this paper provides feedback on existing specifications, mapping failures directly to sentences in the specification language. The procedure described in this paper is implemented within Linear Temporal Logic MissiOn Planning (LTLMoP)[10], [20], 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 [14].The most recent version of LTLMoP can be downloaded online[1]. The LTLMoP implementation of the analysis in this paper is described in [20]; this paper provides the underlying technical details and illustrative examples.

III. Background

In the tasks considered, the robot operates in a known workspace and its behavior (motion and actions) depends on information gathered at runtime from its sensors about events in the environment. We first review some preliminaries, and outline the robot controller synthesis with an example.

Example 1. The example we present is a "fire-fighting" scenario from [10]. The robot task is to enter the house depicted in Fig. 1 from the porch and patrol the rooms. If it encounters a person in the house, the robot should stay in the room with the person and radio for help. If it encounters a potentially hazardous item, the robot should pick it up and take it to the front porch. Furthermore, the robot should continue to patrol the house indefinitely.

A. Linear Temporal Logic

Applying formal methods to continuous problems inherent in robotics requires a discrete abstraction of the problem to enable description with a formal language. We provide details on the discrete abstraction used in this work in Section III-B. The formal language used for high-level specifications in this work is Linear Temporal Logic (LTL) [18]:

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

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. 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; σ hence satisfies the formula $\Box \Diamond \varphi$ if φ is true infinitely often. We refer the reader to [8] for a formal definition of the semantics.

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 [13] into LTL formulas. An excerpt of the user-defined specification for the fire-fighting problem is shown in Listing 1, along with the corresponding LTL formulas for each line. There are two 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 to LTL formulas with operators $\Box$ and $\Diamond$ respectively.

B. Workspace Abstraction

Fig. 1 shows the three stages of workspace abstraction for the "fire-fighting" scenario, from the real environment to a set of convex polygons, and then as a graph with edges connecting adjacent regions. The continuous reactive behavior of a robot is described in terms of a finite set of propositions representing sensor inputs, robot actions and locations. We denote the set of sensor propositions (controlled by the environment) by $\mathcal{X}$, and the set of action and location (i.e., robot-controlled) propositions by $\mathcal{Y}$. In the fire-fighting scenario described above, $\mathcal{X} = \{\pi_{person}, \pi_{hazardous\_item}\}$, $\mathcal{Y} = \{\pi_{porch}, \pi_{deck}, \pi_{bedroom}, \pi_{dining}, \pi_{living}, \pi_{kitchen}, \pi_{radio}, \pi_{pick\_up}, \pi_{drop}, \pi_{carrying\_item}\}$. Propositions π_{person} and $\pi_{hazardous\_item}$ are true when the robot senses a person and a hazardous item respectively, π_{radio} is true when the robot radios for help, π_{pick_up} and π_{drop} are true when the robot picks up and drops a hazardous item respectively, and $\pi_{carrying\_item}$ is true when the robot is carrying a hazardous item.

The value of each proposition can be thought of as the binary output of a low-level black box component (e.g., π_{person} could be set based on a threshold on the output of a sensor, $\pi_{bedroom}$ is set based on a localization component,

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

Listing 1 Example of (an excerpt of) a structured English specification, with corresponding LTL translation

	# Initial conditions	Component of φ_i^e and φ_i^s
1	Env starts with false	$\neg\pi_{person} \wedge \neg\pi_{hazardous\_item}$
2	Robot starts in **porch** with false	$\varphi_{porch} \wedge \neg\pi_{pick\_up} \wedge \neg\pi_{drop} \wedge \neg\pi_{carrying\_item}$
	# Assumptions about the environment	Component of φ_t^e
3	If you were in **porch** then do not **person**	$\square(\varphi_{porch} \Rightarrow \neg \bigcirc \pi_{person})$
4	If you were in **porch** then do not **hazardous_item**	$\square(\varphi_{porch} \rightarrow \neg \bigcirc \pi_{hazardous\_item})$
	# Define robot safety including how to pick up	Component of φ_t^s
5	Do **pick_ up** if and only if you are sensing **hazardous_ item** and you are not activating **carrying_ item**	$\square(\bigcirc \pi_{pick\_up} \Leftrightarrow (\bigcirc \pi_{hazardous\_item} \wedge \neg \bigcirc \pi_{carrying\_item}))$
6	If you did not activate **carrying_ item** then always not **porch**	$\square(\neg\pi_{carrying\_item} \rightarrow \neg \bigcirc \varphi_{porch})$
	# Define when and how to radio	Component of φ_i^s
7	Do **radio** if and only if you are sensing **person**	$\square(\bigcirc \pi_{radio} \Leftrightarrow \bigcirc \pi_{person})$
8	If you are activating **radio** or you were activating **radio** then stay there	$\square((\bigcirc \pi_{radio} \vee \pi_{radio}) \rightarrow \bigwedge_l (\varphi_l \Leftrightarrow \bigcirc \varphi_l))$
	# Patrol goals	Component of φ_g^s
9	If you are not activating **carrying_ item** and you are not activating **radio** then visit **dining**...	$\square \diamondsuit((\neg\pi_{carrying\_item} \wedge \neg\pi_{radio}) \rightarrow \varphi_{dining})$
		...

etc). We define a formula $\varphi_l = \pi_l \bigwedge_{l' \neq l} \neg\pi_{l'}$ to indicate that the robot is in location l and not in any other location (i.e., locations are mutually exclusive). As part of the specification, we encode the possible motion of the robot in the workspace based on the topological constraints of the workspace (i.e., which regions are adjacent, and can be moved between in a single time step). Legal transitions between adjacent regions are represented as edges between vertices in a graph as shown in Fig. 1(c), with implicit self-loops, and then encoded into an LTL formula over location propositions.

LTLMoP takes in a user-defined specification and description of the environment topology, and parses it into a formula of 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, 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 restrictions on the robot's 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 next sensor values as well as the current sensor values and system actions). 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 viewing these formulas as corresponding to system and environment properties, we sometimes refer to φ_i^s and φ_t^e as safety and φ_g^s and φ_g^e as liveness properties.

Given an LTL formula, the synthesis problem consists of constructing an automaton whose behaviors satisfy the formula if such an automaton exists. In general, creating such an automaton is proven to be doubly exponential in the size of the formula. However, by restricting ourselves to the special class of formulas described above, we can use the efficient algorithm introduced in [17], which is polynomial time $O(n^3)$, where n is the number of states. We refer the reader to [17] and [14] for a full description of the synthesis procedure and how it is applied to generate the robot controller. For a description of how the extracted discrete automaton is transformed into low-level robot control, we refer the reader to [14], [10]. However, if the environment can falsify φ_s, we say that the environment is winning and the desired behavior is unsynthesizable.

The fire-fighting specification in Example 1 is synthesizable, and Fig. 2 shows an excerpt of a synthesized automaton; the full automaton has 216 states. Each state 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.

IV. Unsynthesizable Specifications and Undesirable Behavior

Example 2. Recall the synthesizable fire-fighting specification in Listing 1. Consider what happens if we remove the environment safety requirement in line 3, which states that the robot will never see a person when in the porch.

The resulting specification is unsynthesizable, and in particular it is unrealizable, because the environment can force the robot into a safety violation by setting π_{person} to true and $\pi_{hazardous\_item}$ to false in the porch; the robot will radio in the next time step ($\bigcirc \pi_{radio}$ is enforced by line 7), and the system safety then requires it to both stay where it is (i.e., ($\bigcirc \varphi_{porch}$) in line 8), and simultaneously satisfy $\neg \bigcirc \varphi_{porch}$ (line 6) – the system thus has no legal next state.

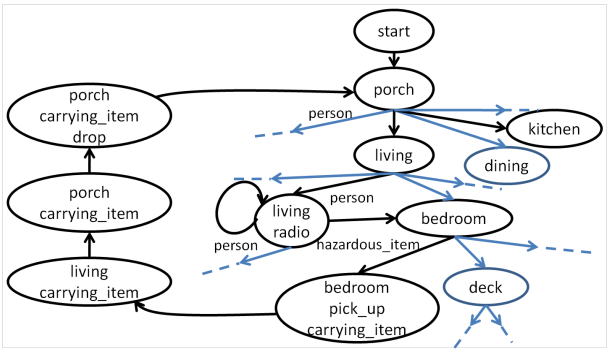


Fig. 2: Excerpt of "fire-fighting" automaton.

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 yet illustrative specification in the fire-fighting 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.

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

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

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

Symmetric to system unrealizability, we can just as well consider winning system strategies that prevent the environment from satisfying the formula φ_e. Overloading terminology, we say that the environment is 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 **person** and do not sense **person**		
	$\square(\bigcirc \pi_{person} \wedge \bigcirc \neg \pi_{person})$	(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 [17] returns a trivial automaton consisting of all the initial states, but no transitions between states. Since the automaton is augmented with self-loops at the controller level, a controller based on this automaton would cause the robot to stay in the start state indefinitely – this is likely not the behavior desired.

We see in the above examples 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 we know which of these is the case, there is further ambiguity surrounding the cause of the unsatisfiability/unrealizability of the specification. In addition, we see that if φ_e is unsatisfiable, then we get a trivial automaton.

B. Causes of Failure

Above we 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-A.1), or if the environment can prevent the system from satisfying one of the liveness conditions (goals) (as in Example IV-A.2). We call the former case *deadlock* and the latter case *livelock*.

C. Identifying Deadlock

We wish to characterize 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 [17]). We can do this using the modal μ-calculus, which extends propositional modal logic with least and greatest fixpoint operators μ, ν [12]. We define the μ-calculus over game structures as in [17]. A formula φ is interpreted as the set of states $[\![\varphi]\!]$ in which φ is true. We also make use of the logical operator $\textcircled{\diamondsuit}$ – informally, a state s is included in $[\![\textcircled{\diamondsuit} \varphi]\!]$ 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 person", then any state in which the system is in the porch would be included in $[\![\textcircled{\diamondsuit} \neg \pi_{person}]\!]$, since the environment cannot activate π_{person} in the next state. Similarly, operator $\textcircled{\square}$ is defined such that a state s is included in $[\![\textcircled{\square} \varphi]\!]$ 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 $\mu X.\ X \vee \textcircled{\square} X$, and constructed by having X initialized to **FALSE** and updated at each iteration with $X \leftarrow X \vee \textcircled{\square} X$ until two iterations are identical. Intuitively,

at each iteration of the fixpoint computation, we add 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 contains an initial state, then the environment can eventually force the system to violate its safety conditions from that initial state, thereby winning the game. Similarly, the system can force the environment into deadlock from states satisfying $\mu X.\ X \vee \textcircled{\bigcirc} X$.

V. Algorithm for Analysis of Specifications

Given a specification, we would like to leverage what we know about the synthesis procedure to determine whether each of φ_e and φ_s is unrealizable or unsatisfiable, and present the user with this information. We assume we are given the input specification parsed into some representation of the environment (φ_e) and robot (φ_s) LTL formulas, and apply a series of tests to determine whether each is unrealizable or unsatisfiable. In LTLMoP, the synthesis algorithm is implemented in the JTLV framework [19], with the corresponding formulas for the initial conditions, transitions and goals represented as Binary Decision Diagrams (BDDs) [15]. The BDD corresponding to a formula is a compact representation of the set of satisfying proposition value combinations, and enables efficient operations on these sets.

In Algorithm 1, AUT_SET is a BDD representating the set of all implementing control automata generated by the synthesis algorithm $SYNTHESIS$ (AUT_SET = **FALSE** if the specification is unsynthesizable). In the BDD representation, **FALSE** denotes the empty set, and **TRUE** denotes the set of all automata. The set of all possible counterstrategies CTR_SET is constructed using the algorithm $COUNTERSTRATEGY$, adapted from [11]. Lines 9-14 check for the unsatisfiability of the initial conditions and the transitions relation (safety) for both environment and system.

- Recall that φ_t^e and φ_t^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, we can perform an emptiness check on the BDD representing the set of variable assignments satisfying $\varphi_i^p \bigwedge_i A_i$ to determine if the transitions in a single time step are satisfiable.
- The above check will not identify unsatisfiability of following the transitions over multiple time steps; for example, the transition relation $\Box(\pi_{pick\ up} \implies \bigcirc \pi_{pick\ up}) \wedge \Box(\pi_{pick\ up} \implies \neg \bigcirc \pi_{pick\ up}) \wedge \Box(\neg\pi_{pick\ up} \implies \bigcirc \pi_{pick\ up})$ is unsatisfiable when starting from the initial condition $\neg\pi_{pick\ up}$, because $\pi_{pick\ up}$ is true in the second time step leading to no valid transitions (since any valid transition would satisfy both $\pi_{pick\ up}$ and $\neg\pi_{pick\ up}$); however our analysis so far will not detect this. For such "multi-step" unsatisfiability of the transitions, we compute the set of environment counterstrategies (which provide sensor inputs such that there is no robot response fulfilling the specification), using a construction similar to [11], and allowing all environments (i.e. setting $\varphi_e =$ `True`). If every sequence of environment moves is in this counterstrategy, then the system must be unsatisfiable.

Algorithm 1 Algorithm for Analyzing a Specification

1: $\varphi_t^p = \bigwedge_j \Box A_j^p, \varphi_g^p = \bigwedge_{i=1}^{n_g^p} \Box\Diamond B_i^p$ for $p \in \{s, e\}$
2: $AUT\_SET \leftarrow SYNTHESIS(s, e)$
3: **if** AUT_SET != **FALSE** (spec. is synthesizable) **then**
4: $\quad AUT \leftarrow AUT\_SET$
5: $\quad$ **if** AUT_SET has no transitions **then**
6: $\quad\quad$ flag as *trivial*
7: **else**
8: $\quad CTR\_SET \leftarrow COUNTERSTRATEGY(s, e)$
9: **if** φ_i^p == **FALSE then**
10: $\quad$ player p has *unsatisfiable initial conditions*
11: **if** $\varphi_i^p \wedge \bigwedge_i A_i^p$ == **FALSE then**
12: $\quad$ p has *unsatisfiable transitions*
13: **if** $\forall\sigma \in CTR\_SET$ (resp. $\forall\sigma \in AUT\_SET$), σ leads to deadlock **then**
14: $\quad$ s (resp. e) *transitions unsatisfiable from initial conditions*
15: **for** $i := 1$ to n_g^p **do**
16: $\quad$ **if** B_i^p == **FALSE then**
17: $\quad\quad$ *p goal i is unsatisfiable*
18: **for** $i := 1$ to n_g^p **do**
19: $\quad$ **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: $\quad\quad$ *p is unsatisfiable between goal i and transitions*
21: **if** $\exists\sigma \in CTR\_SET$ (resp. $\exists\sigma \in AUT\_SET$), σ leads to deadlock) **then**
22: $\quad$ s (resp. e) *unrealizable; can be forced into a safety violation*
23: **for** $i := 1$ to n_g^s **do**
24: $\quad \varphi_g^{s_i} = \bigwedge_{k=1}^{i} \Box\Diamond B_k^s, \varphi_t^{s_i} = \varphi_t^s, \varphi_i^{s_i} = \varphi_i^s$
25: $\quad \varphi_i^{e_i} =$ `True`$, \varphi_t^{e_i} =$ `True`$, \varphi_g^{e_i} =$ `True`
26: $\quad AUT_i \leftarrow SYNTHESIS(s_i, e)$
27: $\quad$ **if** AUT_SET_i == **FALSE** (unsynthesizable) **then**
28: $\quad\quad CTR\_SET_i \leftarrow COUNTERSTRATEGY(s_i, e_i)$
29: $\quad\quad$ **if** CTR_SET_i == **TRUE then**
30: $\quad\quad\quad$ *i^{th} system goal inconsistent with the transition relation*
31: $\quad\quad$ **else if** AUT_SET_{i-1} != **FALSE then**
32: $\quad\quad\quad$ *i^{th} system goal is unrealizable*
33: **for** $i := n_g^e$ to 1 **do**
34: $\quad \varphi_i^{s_i} =$ `True`$, \varphi_t^{s_i} =$ `True`$, \varphi_g^{s_i} = \varphi_g^s$
35: $\quad \varphi_g^{e_i} = \bigwedge_{k=i}^{n_g^e} \Box\Diamond B_k^e, \varphi_t^{e_i} = \varphi_t^e, \varphi_i^{e_i} = \varphi_i^e$
36: $\quad AUT\_SET_i \leftarrow SYNTHESIS(s_i, e_i)$
37: $\quad$ **if** AUT_SET_i != **FALSE** (synthesizable) **then**
38: $\quad\quad$ **if** AUT_SET_i == **TRUE**, AUT_SET_{i+1} != **TRUE then**
39: $\quad\quad\quad$ *i^{th} environment liveness inconsistent with transitions*
40: $\quad\quad$ **else if** AUT_SET_{i+1} == **FALSE then**
41: $\quad\quad\quad$ *i^{th} environment liveness condition is unrealizable*

Moreover, if every sequence of environment moves forces the system into deadlock (rather than livelock), the system safety is unsatisfiable; we identify this (in lines 13-14) with a fixpoint computation as described earlier. The symmetric case for multi-step unsatisfiable environment transitions finds the set of system winning strategies (setting $\varphi_s^i = \varphi_s^t =$ `True`) and checks that every sequence of system actions is winning.

We now turn to checking for unsatisfiability of system and environment liveness in lines 15-20.

- Any liveness condition φ_g^p consists of a conjunction of clauses of the form $\Box\Diamond B_i$, and the safety φ_t^p consists of a conjunction of formulas of the form $\Box A_j$. A contradiction in $\varphi_g^p \wedge \varphi_t^p$ implies that some $\Box\Diamond B_i$ is inconsistent with $\bigwedge_j \Box A_j$, i.e., $\Diamond B_i$ is inconsistent with $\bigwedge_j \Box A_j$. The

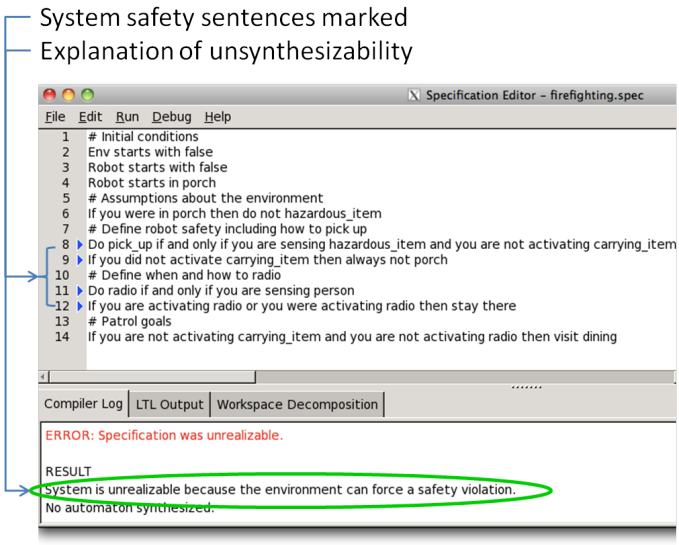


Fig. 3: Analyzing an unsynthesizable specification

A_js in the transition formula govern proposition values in the current and next time steps (as described in Section III-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). We do this 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.

- This check is once again not complete, and to detect multi-step unsatisfiability we look at the counterstrategy/strategy and check that they lead to livelock for the system (lines 29-30) and environment (38-39).

If we do not detect system unsatisfiability, we might still have an unrealizable system specification. To win from an initial state of the game, the environment must either force the system into deadlock or livelock. As described earlier, a reachability analysis using fixpoint operators suffices to check that there is some sequence of environment actions that forces the system into deadlock (lines 21-22), and a symmetric operation checks for the system forcing the environment into deadlock. To illustrate our analysis when the system is unrealizable because of the safety conditions, consider again the fire-fighting specification in Example 2. As mentioned in Section IV, this specification fails to produce a controller, and it is difficult to determine where the problem by inspection alone. Our tests find that the system is unrealizable because the environment can force a safety violation, directing the user's attention to the highlighted sentences (as in Fig. 3).

If the (unrealizable) system cannot be forced into a safety violation, there exists and environment strategy to "lock" the system out of some liveness; we can determine which one by adding the liveness conditions in incrementally to see when synthesis fails (this involves running the synthesizability check once for each liveness condition – see lines 23-32). Additionally, if the environment counterstrategy (allowing all environment transitions) is **TRUE**, then the system is in fact unsatisfiable due to the current set of goals; this test is complete for system unsatisfiability or unrealizability. A symmetric test for the environment runs the synthesis algorithm starting with all environment liveness conditions, allows all system transitions, and removes environment liveness conditions one by one (lines 33-41). If removing an environment liveness condition makes the specification unsynthesizable, the system's winning strategy involved falsifying that liveness, and hence the removed environment liveness was unrealizable (or unsatisfiable if every system strategy is winning after its removal but not before). This however is not a complete test, and there exist cases of environment unrealizability that will not be detected.

In addition to checking for unsatisfiability and unrealizability, trivial automata are easily identified by checking the synthesized automaton for the existence of transitions, as in lines 3-5. By notifying the user prior to execution if the environment is unsatisfiable or unrealizable, we warn them that the behavior generated may not be as intended.

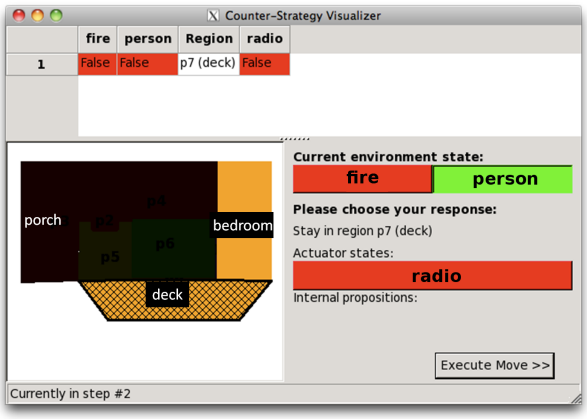
VI. Interactive Exploration of Unrealizable Specifications

In addition to providing explicit feedback on the specification, we provide the user with implicit insights into the failure via an interactive game. We apply a counterstrategy synthesis algorithm similar to that in [11] to extract a strategy for the environment, which finds 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 at 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. 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.

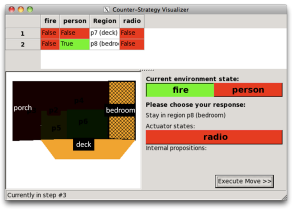
Listing 2 Example of unrealizable specification to be analyzed.

```
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
```

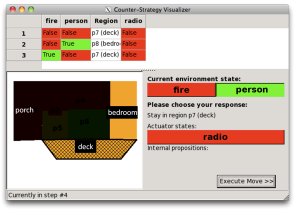
Consider the specification in Listing 2. The environment can prevent the robot from satisfying its goal (to visit the porch infinitely often) by alternately enabling π_{fire} and π_{person}, thereby trapping the robot in the deck and bedroom, i.e. away from the porch. The first three steps of this (infinite) counterstrategy are shown being played through in the Counterstrategy Visualization Tool in Figure 4. 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



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



(b) Robot is moved to $bedroom$; environment turns off $person$ and turns on $fire$, preventing $living$.



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

Fig. 4: Interactive Game

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. 4(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. 4(b)); this prevents the robot from entering the living room, and the user returns to the deck. Now we are back to the original configuration, and the environment again switches on π_{person} and turns off π_{fire}, as shown in Fig. 4(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 video demonstrating the above example is included with this paper.

VII. Conclusions and Future Work

We address the problem of providing automated feedback to the user when a specification for autonomous robot behavior does not have an implementing controller. By exploiting the structure of the specification, we narrow down the possible reasons for failure to create a robot controller, and direct the user's attention to relevant portions of the specification, enabling iterative convergence to a working specification. We also allow the user to explore the cause of failure in an unsynthesizable specification by means of an interactive game. Our approach is implemented as part of the open source LTLMoP toolkit. Future work will leverage existing and novel techniques to further isolate the source of failure and provide the user with additional feedback, including modifications to the input that would enable synthesis of an implementing automaton. If the system or environment is found unsatisfiable, we can further narrow down the cause of this unsatisfiability, by finding an unsatisfiable core. If the system is unrealizable, we can compute the unrealizable core, or add additional environmental assumptions.

References

[1] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. In *Computer Aided Verification*, pages 94–108, 2009.

[2] Amit Bhatia, L. E. Kavraki, and M. Y. Vardi. Sampling-based motion planning with temporal goals. IEEE, 2010.

[3] Roderick Paul Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - a new requirements analysis tool with synthesis. In *Computer Aided Verification*, volume 6174, 2010.

[4] Martin Buehler, Karl Iagnemma, and Sanjiv Singh, editors. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic, George Air Force Base, Victorville, California, USA*.

[5] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *International Conference on Concurrency Theory*, CONCUR '08, 2008.

[6] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. Diagnostic information for realizability. In *Verification, Model Checking, and Abstract Interpretation*, pages 52–67, 2008.

[7] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Stefano Tonetta. Boolean abstraction for temporal logic satisfiability. In *Computer Aided Verification*, pages 532–546, 2007.

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

[9] Georgios E. Fainekos. Revising temporal logic specifications for motion planning. In *IEEE International Conference on Robotics and Automation*, 2011.

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

[11] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design*, pages 152–159, 2009.

[12] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[14] 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.

[15] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:85999, 1959.

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

[17] Nir Piterman and Amir Pnueli. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[18] Amir Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977.

[19] Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. JTLV: A framework for developing verification algorithms. In *Computer Aided Verification*, pages 171–174, 2010.

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

[21] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. In *14th Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, pages 149–167. Springer-Verlag, 2007.

[22] Viktor Schuppan. Towards a notion of unsatisfiable cores for LTL. In *Fundamentals of Software Engineering*, pages 129–145, 2009.

[23] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *ACM International Conference on Hybrid Systems: Computation and Control*, pages 101–110, 2010.