Noname manuscript No.
(will be inserted by the editor)

Automated Generation of Dynamics-Based Runtime Certificates for High-Level Control

Jonathan DeCastro · Rüdiger Ehlers · Matthias Rungger · Ayça Balkan · Hadas Kress-Gazit

Received: date / Accepted: date

Abstract This paper addresses the problem of synthesizing controllers for reactive missions carried out by dynamical systems operating in environments of known physical geometry but consisting of uncontrolled elements that the system must react to at execution time. Such problems have value in semi-structured industrial automation settings, especially those in which robots must behave collaboratively yet safely with their human counterparts. The proposed synthesis framework addresses cases where there exists no satisfying controller for the mission, given the dynamical system and the environment's assumed behaviors. We introduce an approach that leverages information about an abstraction of the dynamical system to automatically generate a concise set of revisions to such specifications. We provide a graphical visualization tool as a design aid, allowing the revisions to be conveyed to the

This work was supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering [grant number CCF-1138996]

J. DeCastro and H. Kress-Gazit
Sibley School of Mechanical and Aerospace Engineering, Cornell University
Ithaca, NY 14853, USA
E-mail: {jad455, hadaskg}@cornell.edu

R. Ehlers
Department of Computer Science, University of Bremen
28359 Bremen, Germany
E-mail: ruediger.ehlers@uni-bremen.de

M. Rungger
Department of Electrical Engineering and Information Technology, Technical University of Munich
80333 Munich, Germany
E-mail: matthias.rungger@tum.de

A. Balkan
Electrical Engineering Department, University of California, Los Angeles
Los Angeles, CA 90095, USA
E-mail: abalkan@ucla.edu

user interactively and added to the specification at the user's discretion. Any accepted statements become certificates that, if satisfied at runtime, provide guarantees for the current mission on the given dynamics. Our approach is cast into a general framework that works with various discrete representations (i.e. abstractions) of the system dynamics. We present case studies that illustrate application of our approach to controller synthesis for two example robotic missions employing different abstractions of the system.

Keywords formal methods · discrete abstractions · counterstrategies · reactive mission planning · dynamical systems

1 Introduction

As industrial processes, homes, and personal vehicles become more automated, formal approaches to mission planning are particularly appealing as a means to creating reliable controllers. Such controllers find utility when complicated tasks must be carried out in collaborative, human environments in which human safety is a primary concern. Automated synthesis of controllers from high-level specifications also frees users from the burdens of directly programming controllers that satisfy complex tasks. Previous works have focused on tools for automatically synthesizing discrete controllers for carrying out high-level mission plans expressed as formal mission specifications, e.g. [25,14,30, 12].

Application of such techniques to physical systems requires the addition of *discrete abstractions* that represent a physical system that must execute the mission (for instance, the dynamics of a mobile robot or a robotic manipulator). Several works have focused attention on computational approaches for obtaining such abstractions that represent a system's dynamics in structured environments; see [11,24,32]. The body of work in computational tools for generating abstractions have enabled others to develop methods for synthesis of mission plans using systems ranging from simple single or double integrators ([10,14]) and piecewise linear models ([27,31]) to nonlinear ([11,2,30,32, 29]), switched ([18]) and hybrid systems ([19]). Synthesis for switched systems was considered in [18,17], where the authors propose methods for computing fine-grained abstractions and switching protocol synthesis for reactive tasks. Tools for automatically synthesizing controllers based on high-level specifications written over task-oriented abstractions of nonlinear systems have been introduced recently in [6].

When there does not exist a controller that is guaranteed to satisfy a specification under the worst-case behaviors of the environment, i.e. it is *unrealizable*, the task of debugging the specification can be difficult and may be worsened with the existence of a discrete abstraction. For instance, if a manipulator tasked with fetching parts on a conveyor belt cannot reach for fast-moving parts, the designer must revise the specification with additional statements that consider both the physics of the manipulator and the underlying assumptions on the uncontrolled environment (in this case, the motion

of the parts). To assist the designer, automated frameworks have been introduced recently for debugging unrealizable specifications [13,22]. Further work has shown progress toward automated repair of unrealizable specifications through the synthesis of revisions to such specifications [9,15,1].

In this paper, we address the problem of realizability of specifications caused by the dynamics of a system by introducing a framework that *automatically* suggests additions to such specifications and provides them to the user in a clear, understandable manner. We focus our work on systems involving physical motion, encompassing tasks carried out by mobile robots, land or air vehicles, or industrial process machinery, to name a few. We adopt an iterative procedure that uses the discrete abstraction of the physical process to assist in interactively computing revisions to the specification. Using a graphical visualization tool, the user may accept or deny the revisions at each step. The goal is to give the user a concise set of revisions to choose from, yet also ones that are consistent with the original intent of the specification. Any revisions that are accepted then become certificates that, if upheld at runtime, will guarantee the mission success.

As an example, consider a collaborative scenario in which a mobile robot tasked with fetching parts in a factory setting, illustrated in Fig. 1. In this scenario, the robot is required to continually visit the supply room and workstations, while avoiding any workstations that are occupied. It must be able to robustly avoid collisions with obstacles and appropriately react to the workstation as it is occupied or unoccupied. If this specification is applied to a vehicle with inertia, the robot's speed and deceleration will become a factor in synthesizing a controller that fulfills the the task. For instance, once a region is sensed as being occupied, the task could fail because the vehicle may not be able to stop by the time it reaches that region, violating the requirement *"avoid any occupied workstations"*. To accommodate the effect of inertia, we could recover if the user is given the environment assumption *"A workstation must not be occupied if the robot is within 1 meter of it,"* and it is accepted for inclusion as an additional assumption in the specification. Notice that, by giving the user the option to accept such assumptions, he/she is aware that the robot will succeed if that assumption is met. On the other hand, the robot may succeed if the assumption is not met, but there are no longer any formal guarantees for the task.

1.1 Related Work

Several researchers have focused attention to the problem of formal synthesis of controllers for physical systems. [2,19] have approached this problem from the standpoint of multi-layered synthesis, where certain parts of the control strategy are left open for an online planner to complete at runtime. Our approach is different in the sense that we seek controllers that guarantee the task under the dynamics at synthesis time, rather than computing a motion plan at runtime. Similar synthesis approaches (e.g. [18,17]) provide guarantees for

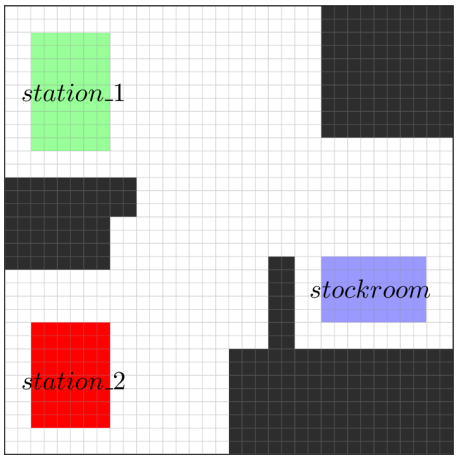


Fig. 1: Factory resupply example scenario.

nonlinear systems, but assume that the specification is realizable. This work is complementary in the sense that we strictly deal with the case of *unrealizability* due to the dynamics of the physical platform. Our approach, moreover, provides the user with a rich source of information regarding compatibility issues with the chosen platform. Specifically, we generate certificates that enable a user to consider the environment's behavior with respect to the mission and the dynamics of the autonomous system.

Our approach for computing revisions is closely related to recent methods described in [9,15,1,16]. In [9], a method is devised for determining the cause of unrealizability for non-reactive tasks and providing specification recommendations to the user. In the reactive setting, [15] present a debugging method for unrealizable specifications based on templates (LTL formulas) mined from an environment counterstrategy. A counterstrategy captures the possible behaviors for the environment for which there are no safe system moves that allow it to fulfill its goals. The method in [1] generates specification templates automatically from the counterstrategy, yielding additional safety and liveness environment statements that remove all execution traces of the counterstrategy. The work of [16] apply the counterstrategy-based environment assumption mining technique to an early warning system in human-in-the-loop control systems, demonstrated in an autonomous driving scenario. By removing the behaviors present in the counterstrategy, the modified environment is restricted in such a way as to permit the system to realize its goals under the strengthened assumptions, but can sometimes lead to specifications that no longer match the user's intent.

The proposed approach differs from existing works in several ways. The closest work to ours, [1], adopt a general approach to specification revisions. Hence, the revisions generated do not hold preference in any one part of a

counterstrategy over any other part, and to avoid placing unnecessary restrictions on either the environment or the behaviors of the system, it is up to the user to decipher which of the generated revisions are important to keep. Our approach, in contrast, proposes formulas that considers the discrete abstraction to guide the creation of environment assumptions that render the specification realizable. The rationale is to propose a concise set of revisions for users to interpret rather than whole counterstrategies. This also aids in managing a large number of revisions.

Another difference from many existing works (e.g. [15,1]) is that we remove the burden for the user to choose templates and subsets of variables for revising specifications. Our algorithm does both automatically, allowing them to simply accept or reject the proposed certificates at each step of an iterative synthesis algorithm. We emphasize how we parse such formulas into statements that are simple to understand, making use of a graphical user interface where applicable.

Recent attention has focused on refining an existing abstraction as a means of rendering a specification realizable. For instance, [20] presents an approach to abstraction refinement in which the authors adopt a counterexample-guided abstraction procedure to iteratively re-partition the state space of a dynamical system, with the goal of satisfying the specification under the dynamical system. Another related work, [7], introduces a partitioning scheme that refines a discrete abstraction of a nonlinear system as a means for repairing unrealizable specifications that are *reactive* in nature. The main distinction is that we reason purely about an uncontrollable environment under a *given* discrete abstraction, so as to characterize the environment behaviors required to guarantee the task under this abstraction. To show the benefit of using our certificates to the task of abstraction refinement with respect to a reactive task, we adopt the state-space-partitioning abstraction refinement procedure of [7] as a case study in the robotics domain.

1.2 Outline

The remainder of this paper is outlined as follows. In Section 2, we review relevant formal definitions and notation. We formally state the problem in Section 3 and present the approach for generating formulas and user feedback in Section 4. In Section 5, we present an approach for parsing the revisions as feedback to the user both as verbal statements and with the help of a graphical tool. Next, we demonstrate our approach for two case studies that employ two different types of discrete abstractions for a mobile robot are presented in Section 6. Lastly, we provide a summary in Section 7.

2 Preliminaries

2.1 Linear Temporal Logic

Linear temporal logic (LTL) formulas are defined over the set AP of atomic propositions in the recursive grammar:

$$\varphi ::= true \mid \pi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where π is an atomic proposition in AP. Respectively, $\wedge$ and $\neg$ are the Boolean operators "conjunction" and "negation", and $\bigcirc$ and U are the temporal operators "next" and "until". From these operators, the following operators are derived: "disjunction" $\vee$, "implication" $\Rightarrow$, "equivalence" $\Leftrightarrow$, "always" $\Box$, and "eventually" $\diamondsuit$.

AP consists of a set of *environment* propositions $\mathcal{X}$ describing the state of sensed environment (e.g. discretized values of a continuous-valued sensor) and a set of *system action* propositions $\mathcal{Y}$ describing the discrete actions the system can take. The LTL formulas are evaluated over infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of truth assignments to the propositions in AP. σ is said to *satisfy* $\bigcirc\varphi$, $\Box\varphi$, or $\diamondsuit\varphi$ if φ holds true in the next position in the sequence, every position, or at some future position(s), respectively. We refer the reader to [28] for a complete definition of the syntax and semantics of LTL formulas.

2.2 Discrete Abstractions

Our model of the behavior of the physical system is a nonlinear differential equation

$$\dot{\xi}(t) = f(\xi(t), \nu(t)) \tag{1}$$

given by the function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$, where $\xi(t)$ is the continuous state of the system and $\nu(t)$ the command input at time $t \in \mathbb{R}_{\geq 0}$. We impose the usual regularity assumptions on f that imply the existence and uniqueness of solutions of (1).

2.2.1 Abstractions of a Dynamical System

Given a bounded configuration space $\mathcal{W} \subset \mathbb{R}^n$, let $\mathcal{R} = \{R_1 \ldots R_p\}$ represent a set of regions (in general, not necessarily disjoint) whose closure covers $\mathcal{W}$, where the open sets $R_i \subseteq \mathcal{W}$. Wherever disjointness must hold, we will state so explicitly. The system (1) may be either open-loop, in which case the inputs ν represent the low-level commands given to the system, or else closed-loop, in which case ν are regarded as high-level commands such as a target region that must eventually be reached under the action of some low-level (feedback) controller that drives the system from one point or region in the state space to another. We define discrete abstractions for both of these cases.

We adopt the motion encoding of [23] by introducing a *completion* proposition $\mathcal{X}_c \subseteq \mathcal{X}$, and let $\mathcal{X}_{nc} = \mathcal{X} \backslash \mathcal{X}_c$ denote those environment propositions not associated with completion (e.g. sensor propositions). Let $\pi_i \in \mathcal{X}_c$ denote a proposition that is `True` iff the system is in a certain configuration or region. Let $\pi_{aj} \in \mathcal{Y}$ denote a proposition that is `True` when the system is *activating* the jth motion command. We assume that all π_{aj} are mutually exclusive – the system can only activate one request at a time.

Definition 1 (*Discrete Abstraction)* We define a *discrete abstraction* S_a as the tuple $(Q_a, V_a, \mathcal{X}_c, \mathcal{Y}, \gamma^a_{\mathcal{X}}, \gamma^a_{\mathcal{Y}}, \delta_a)$, where:

- Q_a is the set of regions discretizing the system's configuration space;
- V_a is the set of discretized system locomotion commands;
- $\mathcal{X}_c$ and $\mathcal{Y}$ are, respectively, the configuration and command propositions (defined above);
- $\gamma^a_{\mathcal{X}} : Q_a \to 2^{\mathcal{X}_c}$ labels each region with the associated proposition in $\mathcal{X}_c$ that evaluates to `True` when the system is in the given region(s);
- $\gamma^a_{\mathcal{Y}} : V_a \to \mathcal{Y}$ labels each discrete command with the associated proposition in $\mathcal{Y}$ that evaluates to `True` when the system is activating the command;
- $\delta_a : Q_a \times V_a \to 2^{Q_a}$ is a nondeterministic transition relation defining a region $\gamma^a_{\mathcal{X}}(q'_a) \in \mathcal{X}_c$ once an action $v_a \in V_a$ is taken when in region $\gamma^a_{\mathcal{X}}(q_a) \in \mathcal{X}_c$, where $q'_a \in \delta_a(q_a, v_a)$.

This abstraction yields an encoding that may be expressed by a set of LTL formulas [23]. Two possible semantics of the abstraction are explained in the case studies in Sec. 6.

2.2.2 Abstractions in the Absence of Dynamics

In the absence of dynamics, let the action propositions $\mathcal{Y}$ represent the workspace regions and $\pi_i \in \mathcal{Y}$ denote a region proposition that evaluates to `True` when the system is in $R_i \in \mathcal{R}$. We consider a topology model, an undirected connectivity graph describing those workspace regions that are accessible and adjacent to one another.

Definition 2 (*Topology Model)* We define a *topology model* as a formula φ^{top}_t over $\mathcal{Y}$ that encodes the allowed next regions given the current region, as follows:

$$\varphi^{top}_t = \bigwedge_{\pi_i \in \mathcal{Y}} \Box \left(\pi_i \implies \bigvee_{\substack{\pi_j \in \mathcal{Y}: \\ cl(R_i) \cap cl(R_j) \neq \varnothing}} \bigcirc \pi_j \right),$$

where $cl(\cdot)$ denotes the closure operation on a set. Note that we also enforce mutual exclusion of regions such that the physical system is only allowed to occupy one region at a time.

2.3 Controller Synthesis

We define a mission specification from which it is required to synthesize a controller for the system.

Definition 3 (*Mission Specifications)* The specifications we consider are LTL formulas of the form:

$$\varphi := \underbrace{(\varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e)}_{\varphi^e} \implies \underbrace{(\varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s)}_{\varphi^s}$$

The formulas φ_i^α, φ_t^α, and φ_g^α are defined over AP, where φ_i^α are formulas for the initial conditions, φ_t^α the allowed transitions (safety conditions) to be satisfied always, φ_g^α the goals (liveness conditions) to be satisfied infinitely often, and $\alpha = \{e, s\}$ (with e for 'environment' and s for 'system'). The liveness guarantees take the form $\bigwedge_{i \in I_\alpha} \square \diamondsuit (B_i^\alpha)$, where I_α is the index set of environment goals B_i^e, defined over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$, or system goals B_i^s, defined over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$. $\mathcal{X}'$ and $\mathcal{Y}'$ are those propositions in $\mathcal{X}$ and $\mathcal{Y}$, respectively, prepended by the $\bigcirc$ operator.

Definition 4 (*Controller Finite State Machine and Execution)* A high-level controller is defined as a finite-state machine (FSM) $\mathcal{A} = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_\mathcal{X}, \gamma_\mathcal{Y})$, where Q is the set of controller states, $Q_0 \subseteq Q$ is the set of initial controller states, $\mathcal{X}$ and $\mathcal{Y}$ are sets of propositions described above, $\delta : Q \times 2^\mathcal{X} \to Q$ is a state transition relation providing the next state $q' \in Q$ given the current state $q \in Q$ and the current value of the environment input $z \in 2^\mathcal{X}$, i.e. $q' = \delta(q, z)$, $\gamma_\mathcal{X} : Q \to 2^\mathcal{X}$ is a labelling function mapping controller states to the set of environment propositions evaluating to `True` for all transitions into that state, and $\gamma_\mathcal{Y} : Q \to 2^\mathcal{Y}$ is a labelling function mapping controller states to the set of action propositions evaluating to `True` in that state.

Consider an infinite *execution* σ of $\mathcal{A}$, where $\sigma = (\gamma_\mathcal{X}(q_0), \gamma_\mathcal{Y}(q_0))(\gamma_\mathcal{X}(q_1), \gamma_\mathcal{Y}(q_1)), \ldots$ for $q_0 \in Q_0$ and $q_i \in Q$, $i > 0$. At each step i in the execution, we say that the environment has made a move (occurring first) if there exists an assignment $\gamma_\mathcal{X}(q_i)$, and that the system has a move (occurring only after the environment has moved) if there exists an assignment $\gamma_\mathcal{Y}(q_i)$. A specification φ written over AP is deemed *realizable* if there exists a finite-state machine $\mathcal{A}$ such that every execution produced by $\mathcal{A}$ satisfies φ. That is, at every $i \geq 0$, there exists an assignment of system variables $\mathcal{Y}$ for all possible assignments of the environment variables $\mathcal{X}$ such that σ satisfies φ. If there exist some environment behaviors on $\mathcal{X}$ for which no such $\mathcal{A}$ can be found, then φ is *unrealizable*.

When combining a mission specification $\varphi^e \implies \varphi^s$ with the topological model, we obtain a *general formula*

$$\varphi := \varphi^e \implies (\varphi^s \wedge \varphi_t^{top}),$$

written over $AP = \mathcal{X} \cup \mathcal{Y}$.

When combining a mission specification with a discrete abstraction, we apply the proposition mapping $\mathcal{X}_{nc} \leftarrow \mathcal{X}$, $\mathcal{X}_c \leftarrow \mathcal{Y}$ and define $\mathcal{Y} = \{\pi_{aj}\}_j$, where each π_{aj} is a robot motion command. We then obtain a *platform-specific formula*

$$\varphi^{abs} := (\varphi^e \wedge \varphi^{e,a}_{t,g}) \implies (\varphi^s \wedge \varphi^{s,a}_t),$$

written over $\mathcal{X}_c \cup \mathcal{X}_{nc} \cup \mathcal{Y}$ in which $\varphi^{e,a}_{t,g}$ consists of environment liveness and safety formulas and $\varphi^{s,a}_t$ consists of system safety formulas, both over $\mathcal{X}_c \cup \mathcal{Y}$, where the superscript a stands for 'abstraction'. Note that φ^{top}_t no longer appears in φ^{abs}. Details on this encoding for specific abstractions presented in Section 6 may be found in [23,5,7].

If φ^{abs} is realizable, then a finite-state machine $\mathcal{A}$ is synthesized by solving a two-player game played between the environment and the system, using a GR(1) synthesis algorithm described in [4].

In the case that φ^{abs} is not realizable, we can synthesize an *environment counterstrategy*: a state machine that captures the possible behaviors for the environment preventing the system from fulfilling its goals.

Definition 5 (*Environment Counterstrategies)* We define an *environment counterstrategy* as a finite-state machine $\mathcal{A}_c = (Q_c, Q_{c0}, \mathcal{X}, \mathcal{Y}, \delta_c, \gamma^c_{\mathcal{X}}, \gamma^c_{\mathcal{Y}})$, where

- Q_c is the set of counterstrategy states;
- $Q_{c0} \subseteq Q_c$ is the set of initial counterstrategy states;
- $\mathcal{X}$, $\mathcal{Y}$ are sets of propositions in AP;
- $\delta_c : Q_c \times 2^{\mathcal{Y}} \to 2^{Q_c}$ is a nondeterministic transition relation returning the set of counterstrategy states at the next position in the sequence given the current state and the current valuations of system commands in $\mathcal{Y}$;
- $\gamma^c_{\mathcal{Y}} : Q_c \to 2^{\mathcal{Y}}$ is a labelling function mapping counterstrategy states to the set of action propositions that are `True` for all transitions into that state, and;
- $\gamma^c_{\mathcal{X}} : Q_c \to 2^{\mathcal{X}}$ is a labelling function mapping counterstrategy states to the set of environment propositions that are `True` in that state.

For notational convenience, we define $\delta^c(q) = \{q' \in Q_c | \exists y_e \in \mathcal{Y} : q' \in \delta_c(q, y_e)\}$ as the projection of $\delta_c(\cdot,\cdot)$ onto the set Q_c, and $\delta^{c^{-1}}(q') = \{q \in Q_c \mid q' \in \delta^c(q)\}$ as the inverse transformation relation mapping counterstrategy states to a set of predecessors.

We now briefly outline how such counterstrategies are synthesized, where the reader is referred to [13] for technical details. Synthesis involves first performing a fixed point computation on a representation of the specification to find a Boolean formula of the winning positions for the environment. From these winning positions, a particular counterstrategy $\mathcal{A}_c$ is extracted. Here, we use the term *positions* to denote assignments over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$ for both the environment and system. We may express the environment's winning

positions as

$$WP_{env} = \mu\mathcal{V}_1. \bigvee_{i_s \in I_s} \nu\mathcal{V}_2. \bigwedge_{i_e \in I_e} \mu\mathcal{V}_3. \left(\neg B^s_{i_s} \vee \mathsf{Pre}\mathcal{V}_1\right) \wedge \mathsf{Pre}\mathcal{V}_2 \wedge \left(B^e_{i_e} \vee \mathsf{Pre}\mathcal{V}_3\right),$$

where μ and ν are, respectively, the least and greatest fixpoint operators, and where $\mathcal{V}_1$ and $\mathcal{V}_3$ are initially `True`, and $\mathcal{V}_2$ is initially `False` before the fixed point computation begins. $\mathsf{Pre}\mathcal{V}$ is an enforceable predecessor operator that produces, for a set of positions $\mathcal{V}$, a set of predecessor positions such that, there exists an environment assignment satisfying the environment safety formulas φ^e_t and $\varphi^{e,a}_t$, for all possible action assignments satisfying the system safety formulas φ^s_t and $\varphi^{s,a}_t$.

3 Problem Formulation

Assume the original mission specification φ under the assumption of a topological model φ^{top}_t is *realizable*, where the formulas φ^e and φ^s are, respectively, environment assumptions and system guarantees defined by the user. Additionally, we require that φ^e is not falsified by system behaviors satisfying φ^s (i.e. no trivial behaviors). The goal is to synthesize controllers for such specifications. Given the discrete abstraction and the formulas φ^e and φ^s, we seek a controller for the platform-specific specification φ^{abs}.

If φ is realizable and φ^{abs} is *unrealizable*, then, according to Definition 4, there exists some environment behavior in $\mathcal{X}_{nc}$ admissible by φ^e such that, if behaviors in $\mathcal{X}_c$ satisfy $\varphi^{s,e}_{t,g}$, then no system behaviors satisfy $\varphi^s \wedge \varphi^{s,a}_t$.

In the case that φ^{abs} is *unrealizable*, the goal of this paper is to generate a set of revisions (LTL formulas) that, upon conjunction with the platform-specific formula φ^{abs}, render it realizable. Specifically:

Problem 1 (*Revision Generation and User Feedback)* Given φ realizable, φ^{abs} unrealizable, automatically derive a set of formulas for the environment and the system such that

$$\varphi^{mod} := (\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t) \implies (\varphi^s \wedge \varphi^{s,a}_t \wedge \psi^s_t) \tag{2}$$

is realizable and both $\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t$ and $\varphi^s \wedge \varphi^{s,a}_t \wedge \psi^s_t$ are satisfiable. For any such revisions, provide the user with a suggested set of *runtime certificates*: a set of human-readable statements consisting of safety assumptions and guarantees, resp. ψ^e_t and ψ^s_t, and liveness assumptions ψ^e_g.

To illustrate the problem, consider the following example.

Example 1 Consider again the factory setting of Fig. 1. We begin by writing the specification in terms of a topology model. Under such a model, the regions of the workspace to be visited are encoded as actions, hence *stockroom*, *station_1* and *station_2* belong to $\mathcal{Y}$, while *s1_occupied* and *s2_occupied* (indicating when the respective region is occupied) are sensors belonging to $\mathcal{X}$.

The robot is required to visit the stockroom and the two workstations (system liveness) but avoid visiting those that are occupied (system safety). If the robot is within a workstation, the environment is required to keep that station unoccupied (environment safety). Also, the workstations are required to be infinitely often unoccupied (environment liveness).

The general specification φ is composed of the following formulas:

$$\begin{aligned}
&\square\diamondsuit \mathit{stockroom} \wedge \square\diamondsuit \mathit{station_1} \wedge \square\diamondsuit \mathit{station_2} && \triangleleft \text{ sys liveness}\\
&\square\diamondsuit \neg \mathit{s1_occupied} \wedge \square\diamondsuit \neg \mathit{s2_occupied} && \triangleleft \text{ env liveness}\\
&\square(\bigcirc \mathit{s1_occupied} \implies \bigcirc \neg \mathit{station_1}) && \triangleleft \text{ sys safety}\\
&\square(\bigcirc \mathit{s2_occupied} \implies \bigcirc \neg \mathit{station_2}) && \triangleleft \text{ sys safety}\\
&\square(\mathit{station_1} \implies \bigcirc \neg \mathit{s1_occupied}) && \triangleleft \text{ env safety}\\
&\square(\mathit{station_2} \implies \bigcirc \neg \mathit{s2_occupied}) && \triangleleft \text{ env safety}\\
&\texttt{True} && \triangleleft \text{ sys init}\\
&\texttt{True} && \triangleleft \text{ env init}
\end{aligned}$$

and the following topology model:

$$\begin{aligned}
\phi_t^{top} = &\square\,(\mathit{stockroom} \implies \bigcirc\, \mathit{factory_floor} \bigcirc\, \mathit{stockroom}) \wedge\\
&\square\,(\mathit{station_1} \implies \bigcirc\, \mathit{factory_floor} \vee \bigcirc\, \mathit{station_1}) \wedge\\
&\square\,(\mathit{station_2} \implies \bigcirc\, \mathit{factory_floor} \vee \bigcirc\, \mathit{station_2}) \wedge\\
&\square\,(\mathit{factory_floor} \implies \bigcirc\, \mathit{stockroom} \vee \bigcirc\, \mathit{station_1} \vee\\
&\qquad\qquad \bigcirc\, \mathit{station_2} \vee \bigcirc\, \mathit{factory_floor})\,.
\end{aligned}$$

where $\mathit{factory\_floor}$ corresponds to the unlabeled white space in Figure 1. The specification φ is realizable.

Now suppose we are given a fully-actuated planar robot governed by inertia, described by the system

$$\ddot{x} = u \qquad \ddot{y} = v, \tag{3}$$

where $(u, v) \in U$ are robot commands and $(x, y) \in \mathbb{R}^2$ are the Cartesian robot coordinates. We derive an abstraction S_a of these dynamics in the configuration space $(x, y, \dot{x}, \dot{y})^T \in \mathcal{W}$ and obtain the formula φ^{abs}. With S_a, we want to synthesize a realizable controller for φ^{abs} that satisfies the task given an abstraction of these dynamics.

The specification may be unrealizable for a number of different reasons. One cause is *deadlock*, where the environment can force the system into certain states that have no legal transitions. Suppose that the robot has inertia, which is encoded in the abstraction as requiring two regions before it is able to decelerate to a stop. If $\mathit{s1\_occupied}$ turns from `False` to `True` when the robot is within two grid cells of $\mathit{station\_1}$, it will be unable to avoid a collision. Note that, as this behavior stems from the robot's physics, this behavior occurs in the platform-specific specification but not in the general specification.

Another cause of unrealizability is due to *livelock*, in which the system is prevented from reaching its goals as a result of an infinite sequence of environment inputs. For instance, suppose the robot is approaching *station*_1 and the environment toggles the value of *s*1_*occupied* infinitely often. If the switching is fast enough, the robot may be able to change its heading, but unable to move forward toward the workstation. The behavior does not appear in the general formula because the topology graph always allows the robot to either remain in place or transit to an adjacent region once the environment has made a move.

4 Revising Unrealizable Specifications via Counterstrategies

In this section, we formalize our solution strategy for Problem 1. The goal is to generate revisions that, if possible, repair unrealizable specifications where the unrealizability is a consequence of a discrete abstraction included in the specification. We adopt an iterative approach that takes a specification φ and dynamical system f, and interacts with the user at various stages of the process, as illustrated in Fig. 2. If successful, the approach outputs a finite state machine $\mathcal{A}$ and the user is exposed to the revisions that have been added to the specification. At each step, the user may choose to accept the revisions or else supply their own handwritten revisions. The interactive nature of the algorithm allows the specification designer to choose revisions that are consistent with his or her intent.

The first step is to create an abstraction of the specification (Abstraction in Fig. 2), either via a gridding procedure [5] or via the constructive procedure giving rise to partitions based on reachability analysis [6]. In the latter case, Abstraction contains a call to synthesize a finite-state machine from φ, upon which atomic controllers may be constructed. If the specification φ^{abs} or the modified specification φ^{mod} is unrealizable, various stages of the revisions approach are invoked as necessary; the complete process is discussed in this section. The Realizable blocks take as input a specification and returns a controller FSM $\mathcal{A}$ if realizable; otherwise, a counterstrategy $\mathcal{A}_c$ is returned. $\mathcal{A}_c$ will differ depending on whether it is synthesized from a deadlock-modified specification or livelock-modified specification. Note that, if the specification is unrealizable and the counterstrategy is the same between iterations of the while loop, this means that no revisions have been found that meet the user's criteria or do not falsify the specification. In this case, the algorithm terminates with an *unrealizable* output.

Note that there are many potential counterstrategies for a given unrealizable specification, with each one being assembled from a *game structure* [4] and contain a subset of behaviors of the game structure. When extracting revisions to the specification, we reason on counterstrategies rather than on game structures for two reasons. First, game structures have at least as many behaviors as a counterstrategy, so an approach that extracts revisions on a game structure will produce at least as many revisions as one applied to a

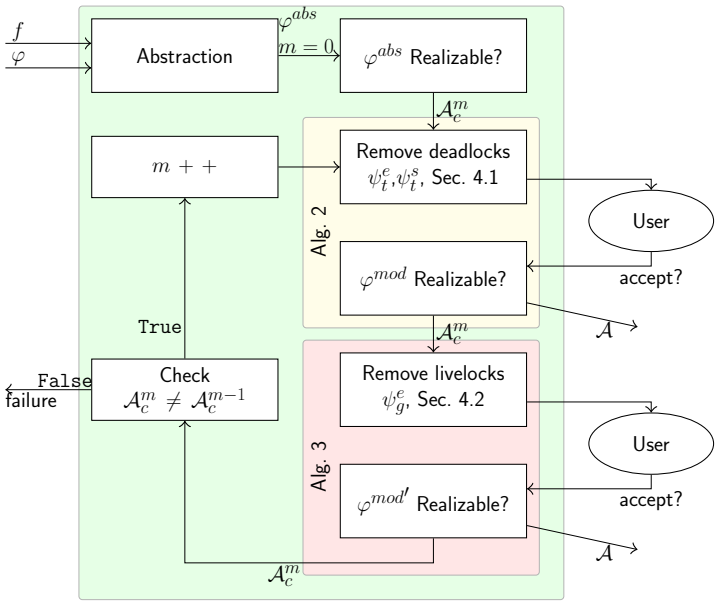


Fig. 2: Overview of the procedure for finding runtime certificates and controller synthesis.

counterstrategy of that game structure. Having fewer revisions in general reduces the conservatism and lessens the number of assumptions that are fed to the user. Second, there are many possible counterstategies for a given game structure to draw from. Using existing counterstrategy-synthesis tools (e.g. [3, 8]), these can be readily customized to suit a particular designer's needs. For instance, a counterstrategy could be extracted that minimizes some objective function (e.g. minimizes distance to a goal) or that gives preference to certain states over others (e.g. wide passageways over narrow ones). Thus, our aim is to introduce an approach that generates a small number of revisions, and may be customized to a user's design intent.

We introduce the following example to illustrate the major concepts discussed in the remainder of this section.

Example 2 Consider the workspace shown in Fig. 3a. Given $\mathcal{X} = \{sen\}$ where sen is the sensor input and $\mathcal{Y} = \{r1, r2\}$, we write a specification φ requiring the robot to visit $r2$ (lower-left gray region) when sen is `False`, but avoid $r2$

when *sen* is `True`. Formally:

$$\begin{aligned}
&\square\diamondsuit r2 && \triangleleft\ \varphi_g^s\\
&\square\diamondsuit \neg sen && \triangleleft\ \varphi_g^e\\
&\square(\bigcirc sen \implies \bigcirc \neg r2) && \triangleleft\ \varphi_t^s\\
&\square(r2 \implies \bigcirc \neg sen) && \triangleleft\ \varphi_t^e\\
&\texttt{True} && \triangleleft\ \varphi_i^s\\
&\texttt{True} && \triangleleft\ \varphi_i^e
\end{aligned}$$

The initial conditions are denoted `True` to signify that an execution can begin with any safe initial robot configuration, action, and environment settings. The controller satisfying this specification is given in Fig. 3b.

We are now given an abstraction, automatically derived using the procedure in [23], in which we have redefined the complete set of environment variables to be $\mathcal{X} = \mathcal{X}_c \cup \mathcal{X}_{nc}$, where we map the set of environment propositions in φ to the set $\mathcal{X}_{nc}$ (in this case $\mathcal{X}_{nc} = \{sen\}$). The set $\mathcal{X}_c = \{x1, \ldots, x16\}$ is an encoding of the set of 2-D robot configurations, and we replace the set of robot actions $\mathcal{Y}$ with the robot's four cardinal directions of motion. An excerpt of two of the conjuncts in φ^{abs} for which the robot's current position is $x7$ is as follows:

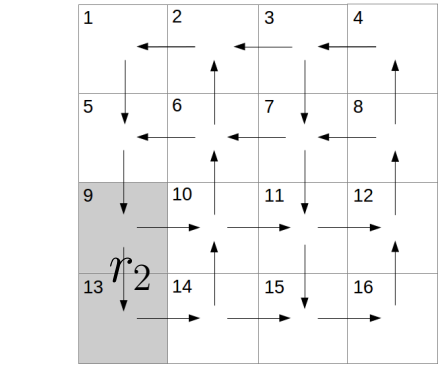
$$\begin{aligned}
&\square((x7 \wedge W) \implies (\bigcirc x6 \vee \bigcirc x7)) \wedge \square((x7 \wedge S) \implies (\bigcirc x11 \vee \bigcirc x7)) && \triangleleft\ \varphi_t^s,\\
&\square\diamondsuit((x7 \wedge W) \implies \bigcirc x6) \wedge \square\diamondsuit((x7 \wedge S) \implies \bigcirc x11) && \triangleleft\ \varphi_g^e.
\end{aligned}$$

Additionally, φ^{abs} includes conjuncts in φ_t^s and φ_t^e that encode, respectively, mutual exclusion of action and completion propositions (no two actions or completions may occur at the same time). The complete discrete abstraction S_a appears as arrows in Fig. 3a. A new specification φ^{abs} is derived, where:

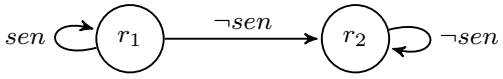
$$\begin{aligned}
&\square\diamondsuit(x9 \vee x13) && \triangleleft\ \varphi_g^s\\
&\square\diamondsuit \neg sen && \triangleleft\ \varphi_g^e\\
&\square(\bigcirc sen \implies \bigcirc \neg(x9 \vee x13)) && \triangleleft\ \varphi_t^s\\
&\square((x9 \vee x13) \implies \bigcirc \neg sen) && \triangleleft\ \varphi_t^e\\
&\texttt{True} && \triangleleft\ \varphi_i^s\\
&\texttt{True} && \triangleleft\ \varphi_i^e
\end{aligned}$$

4.1 Preventing Deadlock

In preventing deadlock, we introduce a scheme to process a counterstrategy and extract a set of environment assumptions that remove deadlock behaviors.



(a)



(b)

Fig. 3: 2-D example. (a) shows the workspace map and grid whose cells are labeled with the configuration variable. The white grid cells denote r_1, while the gray denote r_2. (b) shows the synthesized controller for φ.

Consider a counterstrategy $\mathcal{A}^c$ whose deadlock states are collected in $Q_{dead} = \{q \in Q_c \mid \delta^c(q) = \varnothing\}$, and let

$$B_{robot}(q) = \bigwedge_{\pi \in \gamma^c_{\mathcal{Y}}(q) \cup (\mathcal{X}_c \cap \gamma^c_{\mathcal{X}}(q))} \pi \wedge \bigwedge_{\pi \in (\mathcal{Y} \cup \mathcal{X}_c) \setminus (\gamma^c_{\mathcal{Y}}(q) \cup \gamma^c_{\mathcal{X}}(q))} \neg\pi,$$

$$B_{env}(q) = \bigwedge_{\pi \in \mathcal{X}_{nc} \cap \gamma^c_{\mathcal{X}}(q)} \pi \wedge \bigwedge_{\pi \in \mathcal{X}_{nc} \setminus \gamma^c_{\mathcal{X}}(q)} \neg\pi.$$

In words, $B_{robot}(q)$ denotes a Boolean formula for the truth-values of the command and configuration propositions $\mathcal{Y} \cup \mathcal{X}_c$ at state q and $B_{env}(q)$ denotes a Boolean formula for the truth-values of the subset of environment input propositions $\mathcal{X}_{nc}$ at state q.

4.1.1 Removing Deadlock

As in [1], any pair of states $q^i, q^j \in Q_c$ in a counterstrategy satisfying $q^j \in \delta^{c^{-1}}(q^i)$ can be expressed with the formula

$$\bigvee_{q^j \in \delta^{c^{-1}}(q^i)} \Diamond \left(B_{robot}(q^j) \wedge \bigcirc B_{env}(q^i) \right). \tag{4}$$

For a particular q^i, the formula characterizes the environment's behavior at this state in the counterstrategy and the robot's configuration at the state immediately prior. If there exists an execution that eventually reaches a deadlock

state $q^i_{dead} \in Q_{dead}$, we may use this statement to remove the environment behaviors in the counterstrategy causing deadlock. The negation of (4) leads to the following formula, which is suggested as an additional assumption:

$$\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \square \left(B_{robot}(q^j) \implies \bigcirc \neg B_{env}(q^i_{dead}) \right). \tag{5}$$

Before conjuncting each computed formula with ψ^e_t, a check is made to determine if it falsifies the left-hand side of φ^{mod}, i.e. if (5) is assigned to $\varphi^{e,a}_t$, then $\varphi^e \wedge \varphi^{e,a}_t \wedge \psi^e_t = \texttt{False}$. If this is the case, the formula is discarded and it is not included as a conjunct in ψ^e_t. The deadlock state index i is then incremented and the process repeats with a new suggested assumption.

4.1.2 Preventing Unintended Behaviors

When the environment assumptions of (5) are added to the specification, the system may exhibit behaviors that are noticeably different from the system's behaviors from the original specification synthesized under a topology graph. For instance, consider Example 2. Under the original (realizable) specification φ, if the robot is in the white region in Fig. 3a and the door is closed, the FSM of Fig. 3b reveals that the system remains within the white region until the door opens. Now consider the platform-specific specification φ^{abs} abstracted under the shown workspace decomposition. After applying assumptions to prevent the door from closing when the system is in cell $x5$ (where the system is unable to avoid entering r_2 in the next step), the system will not avoid moving toward the door when within the white region and the door is closed, clearly different behavior than that in Fig. 3b.

To reconcile these differences, our approach forces the system to react conservatively to the newly-added environment revision by treating the state $q^j \in \delta^{c^{-1}}(q^i_{dead})$ (in the antecedent of (5)) as if it were a deadlock state. Thus, when the physical system is at a configuration previous to the deadlocked configuration and when the added assumptions on the environment prevent it from behaving in a certain way in the next state, it will be forbidden from entering the configuration prior to deadlock when these conditions hold in the current state. In other words, such conditions will prevent the system from approaching the door when it is closed. On the other hand, when the door is open, ψ^e_t prevents it from closing again once the system has made its move.

Formally, we disallow the behavior

$$\bigvee_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \diamondsuit (\bigcirc B_{env}(q^i_{dead}) \wedge \bigcirc B_{robot}(q^j))$$

by introducing an additional revision ψ^s_t:

$$\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \square \left(\bigcirc B_{env}(q^i_{dead}) \implies \bigcirc \neg B_{robot}(q^j) \right). \tag{6}$$

Such a revision places a safety restriction on the system, preventing it from entering a *neighboring* state to a deadlock state whenever the environment is set to the same value for which deadlock occurs. Doing this produces a specification that makes the system's behavior conservative; we are limiting the conditions under which the system may enter a neighboring state, when in fact the system is not in any true danger of violating the original safety guarantees in φ until it reaches r_2. Nonetheless, if the specification is realizable, the system will be able to react to the environment as long as the actions/configurations are not included in those specified in $\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} B_{robot}(q^j)$.

4.1.3 Aggregated Deadlock Removal via Backward Reachability

If the modified formula is determined to be unrealizable and new deadlock states are found at a state $q^j \in \delta^{c^{-1}}(q^i_{dead})$, then we once again return to the original set of circumstances specified in Problem 1. We repeat the process in this section for as many times as required to eliminate deadlock states or until the resulting specification is unrealizable. This, however, has the drawback that synthesis of a counterstrategy may have to be repeated several times. We adopt a more direct approach that reduces the number of computations, and has the added benefit of producing user-generated statements that are simple to interpret (see Sec. 5).

To avoid repeated synthesis of counterstrategies, we apply the assumption and guarantee revisions explained above to entire *subtraces* of a single counterstrategy (a finite word of an execution trace for the counterstrategy). To do this, we identify states for which there is no safe command to be taken such that there exists a subtrace that eventually visits states in $Q_c \backslash Q_{dead}$. The search for deadlock revisions then reduces to a graph search on the counterstrategy, as summarized in Algorithm 1. The algorithm builds up a set of *deadlock-committed* states Q_{commit} by querying the abstraction and adding, via a breadth-first search (BFS in line 4), predecessor counterstrategy states from deadlock Q_{dead} for which all system commands lead to states in Q_{commit}. As such, the procedure BFS amounts to a backward reachability operation.

For generating revisions and providing user feedback, we also maintain a mapping $Q_{reach} : Q_{commit} \to 2^{Q_{dead}}$ of deadlock states reachable from each $q \in Q_{commit}$ obtained from the abstraction S_a. The search continues until a fixed point of states is reached where no additional deadlock-committed states can be found, at which point BFS returns a tuple containing Q_{commit} and Q_{reach}. The precise condition under which the search terminates is when a $q \in Q_c$ is found such that:

$$\exists q' \in \delta^c(q) : q' \notin Q_{commit}.$$

Here, Q_{commit} plays the role of Q_{dead}. We therefore replace Q_{dead} in the safety revisions (5) and (6) with Q_{commit}. To be precise, we replace (5) and (6) with,

respectively:

$$\bigwedge_{q^j \in Q^i_{commit}} \Box \left(B_{robot}(q^j) \implies \bigwedge_{q^k \in Q_{reach}(q^i_{commit})} \bigcirc \neg B_{env}(q^k) \right) \tag{7}$$

$$\bigwedge_{q^j \in Q^i_{commit}} \Box \left(\bigwedge_{q^k \in Q_{reach}(q^i_{commit})} \left(\bigcirc B_{env}(q^k) \implies \bigcirc \neg B_{robot}(q^j) \right) \right), \tag{8}$$

for each $q^i_{commit} \in Q_{commit}$.

Algorithm 2 contains a procedure for finding revisions that target deadlocks. The first step in the algorithm is to compute environment and system transition subformulas ψ^e_t and ψ^s_t (using the approach described in Sec. 4.1.3) that prevent transitions to states in the counterstrategy from which the system has no safe transitions (deadlock). If these revisions falsify the environment and system, they are removed. The second step is to provide feedback to the user. Depending on the user's response, the revisions are either applied or discarded. If accepted, they become runtime certificates, as discussed in Sec. 5.

Algorithm 1 Computing deadlock-committed states.

```
    procedure COMMITSTATES(Q_dead)
        Initialize Q_new, Q_commit to Q_dead.
        while Q_new ≠ ∅ do
            Q_new ← BFS(A_c, Q_commit)
5:          Q_commit ← Q_commit ∪ Q_new
            for q ∈ Q_new do
                Q_reach(q) ← δ^c(q)          ▷ Create a graph of reachable states
            end for
        end while
10:     return Q_commit, Q_reach
    end procedure
```

The following proposition is immediate considering the fact that Algorithm 1 traverses a particular counterstrategy $\mathcal{A}_c$ via backward reachability using the transition system S_a.

Proposition 1 *Algorithm 1 is sound and complete with respect to S_a.*

Soundness with respect to S_a implies that Q_{reach} does not contain states for which there exists no sequence of actions that may eventually lead to Q_{dead}. *Completeness with respect to S_a* implies that, from each state in Q_{reach}, there exists a sequence of actions that may eventually reach Q_{dead} and Algorithm 1 will always terminate with such a Q_{reach}, if it exists.

Example 3 Returning to Example 2, suppose we obtain a counterstrategy containing the states $q_0, \ldots, q_3$, as pictured in Fig. 4a and Fig. 4b, starting in cell $x7$ with $sen =$`False`. One of the possible executions in this counterstrategy

Algorithm 2 Synthesizing deadlock revisions for an realizable specification φ^{abs}.

procedure SYNTHDEADLOCKREVISIONS(φ^{abs},$\mathcal{A}_c$,$\mathcal{R}$)

▷ *Eliminate deadlocks*

$Q_{commit} \leftarrow \mathsf{commitStates}(\mathcal{A}_c)$
for all $q^i_{commit} \in Q_{commit}$ **do**
5: $\psi^e_{t,cand}, \psi^e_t \leftarrow$ Eq. (7)
$\psi^s_{t,cand}, \psi^s_t \leftarrow$ Eq. (8)
if $\neg(\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t \wedge \psi^e_{t,cand})$ or $\neg(\varphi^s \wedge \varphi^{s,a}_t \wedge \psi^s_t \wedge \psi^s_{t,cand})$ **then**
$\psi^e_t \leftarrow \psi^e_t \backslash \psi^e_{t,cand}$
$\psi^s_t \leftarrow \psi^s_t \backslash \psi^s_{t,cand}$
10: **end if**
end for

▷ *User feedback*

for all $R_i \in \mathcal{R}$ **do**
$(q^\star_i, q^\star_{dead}) \leftarrow$ Eq. (22)
15: $dist_i \leftarrow |\gamma^c_{\mathcal{X}}(q^\star) - \gamma^c_{\mathcal{X}}(q^\star_{dead})|$
print $(R_i, dist_i, B_{env}(q^\star_{dead,i}))$
end for
if user accepts any ψ^e_t, ψ^s_t **then**
$\varphi^{mod} \leftarrow$ Eq. (2)
20: $(realiz, \mathcal{A}^m_c, WP_{env}) \leftarrow \mathsf{ctrStrategy}(\varphi^{mod})$
end if
return $realiz$,$\mathcal{A}_c$,WP_{env},φ^{mod}
end procedure

eventually leads the robot to cell $x5$ with the sensor sen =`True` as shown in Fig. 4b. In this execution, the sensor sen remains `False` until the robot enters $x5$, at which point a transition in φ^s_t is violated. Hence q_3 is a deadlock state. The formula $\diamondsuit(\bigcirc sen \wedge \neg x6 \wedge W)$[1] (in words, "eventually, the system will be in cell 6 and activating go West, with sen `True` in the following time step") is extracted by evaluating $\diamondsuit(B_{robot}(q_2) \wedge \bigcirc B_{env}(q_3))$. The complement of this formula, $\square((\neg x6 \wedge W) \implies \bigcirc \neg sen)$, is added as an additional environment assumption. This assumption negates the behaviors in the counterstrategy for that particular deadlock state. Being that there is only one deadlock state, we add no further assumptions. Upon adding this revision to the environment assumptions, we determine that the modified specification is realizable.

Notice that the controller synthesized based on the specification above produces executions that satisfy the specification but the system now assumes that the environment will always turn sen `False` whenever it reaches $x5$. In the example, consider the behavior when the robot starts at $x7$ with sen `True`. The execution of the robot in this case is as shown in Fig. 4c. In this execution, sen remains `True` and, as the robot moves toward $r2$, the environment eventually must set sen to `False` to be consistent with the added assumption. When outside of $x5$, the robot follows the same sequence of moves regardless of the environment. Note that the controller for the original, realizable specification

[1] We only make the `True` action explicit (W in this case), since mutual exclusion disallows the other actions from being activated at the same time.

φ (Fig. 3b) does not exhibit this behavior because there is no imposition on how the environment must behave based on the robot's configuration. In that case, if *sen* is `True`, the robot waits in $r1$ until *sen* becomes `False`.

We compute a set of four deadlock-commit states $Q_{commit} = \{q'_1, q'_2, q'_3, q'_4\}$ corresponding to the cells $\{x5, x1, x2, x6\}$. We obtain the following ψ_t^e formulas:

$$\square((x5 \wedge S) \implies \bigcirc \neg sen) \tag{9}$$
$$\square((x1 \wedge S) \implies \bigcirc \neg sen) \tag{10}$$
$$\square((x2 \wedge W) \implies \bigcirc \neg sen) \tag{11}$$
$$\square((x6 \wedge N) \implies \bigcirc \neg sen), \tag{12}$$

and the following ψ_t^s formulas:

$$\square(\bigcirc sen \implies \bigcirc \neg(x5 \wedge S)) \tag{13}$$
$$\square(\bigcirc sen \implies \bigcirc \neg(x1 \wedge S)) \tag{14}$$
$$\square(\bigcirc sen \implies \bigcirc \neg(x2 \wedge W)) \tag{15}$$
$$\square(\bigcirc sen \implies \bigcirc \neg(x6 \wedge N)). \tag{16}$$

With these revisions added to ψ_t^e and ψ_t^s (highlighted orange in Fig. 5, the modified specification eliminates the deadlock states present in the original counterstrategy. Observe that (9) alone will eliminate deadlock at cell $x5$; however, (9) coupled with the system safety formulas (13) will introduce another deadlock at $x1$ and $x6$, and so on. Hence, (9) – (16) are necessary when *both* environment and system safety formulas (7) and (8) are added at each state in Q_{commit}. Additionally, note that none of the revisions falsify the environment.

Upon synthesis, we find that a counterstrategy synthesized from this modified specification does not contain deadlock states. In the next section, we discuss an approach to render the specification realizable through an elimination of livelock behaviors.

4.2 Preventing Livelock

The environment may be able to win the two-player game through livelock: moves for the environment that force the system to cycle indefinitely through a sequence of states, keeping the system away from one of its goals. Consider the behavior of the system when the above ψ_t^e and ψ_t^s formulas (9)–(16) are introduced as revisions. Starting at $x16$, the behavior shown in Fig. 5 is possible. In this execution (shown in Fig. 5), the system eventually cycles indefinitely between six cells in the workspace. Whenever the robot visits the cell $x7$, the environment activates *sen*, forcing the robot to move S to avoid violating the safety guarantee revision in (16). The environment is then able to satisfy its liveness goal ($\square\diamondsuit(\neg sen)$), while preventing the system from achieving its goal of reaching r_2.

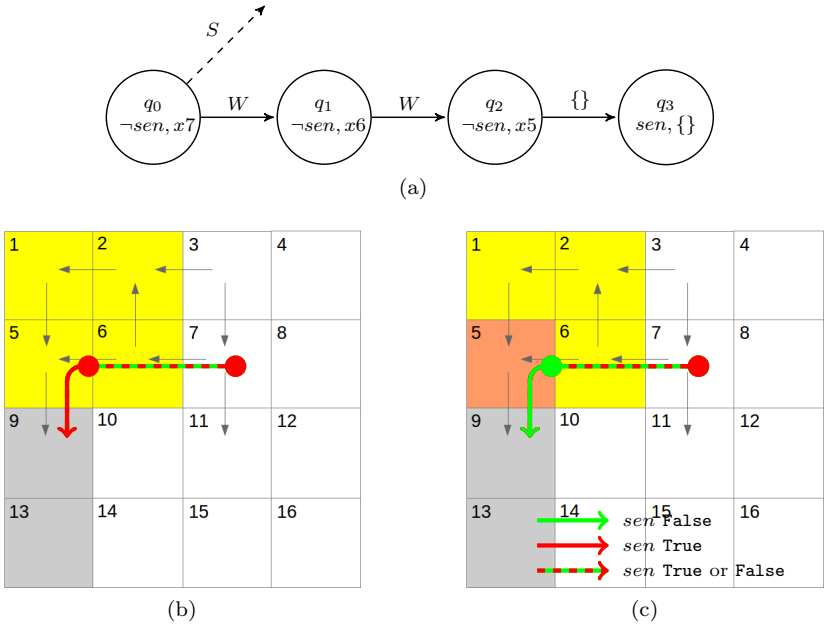


Fig. 4: (a) shows a partial counterstrategy for Example 3 leading to deadlock. (b) shows a corresponding robot trajectory leading to deadlock. The cells shaded yellow indicate configurations in which there are no sequence of commands that avoid reaching r_2 eventually. (c) shows the result of a synthesized controller where deadlock is removed, but where the strategy *expects* the environment to set *sen* to `False` once the robot enters cell $x5$. The numbering of the cells correspond to the state labels, omitting the "x".

Once we obtain a counterstrategy free of deadlock states, our approach generates environment assumptions that remove the counterstrategy executions that exhibit livelock. The idea is to selectively identifies states in the counterstrategy for which the system still has winning actions to take. Our approach then exploits states with this property to prevent the environment from always making such assignments at these counterstrategy states. We do so by applying liveness assumptions that remove the environment's ability to remain in these states forever. Hence, there exists some finite time in the execution where the system is allowed to take these actions.

Take any $q \in Q_c$. Let $V_{nc} : Q_c \to 2^{\mathcal{X}_{nc}}$ be a function mapping counterstrategy states to the set of all non-completion environment proposition assignments at the current step in the execution that satisfy the environment's safety formula φ_t^e, with respect to the proposition assignments at the *previous*

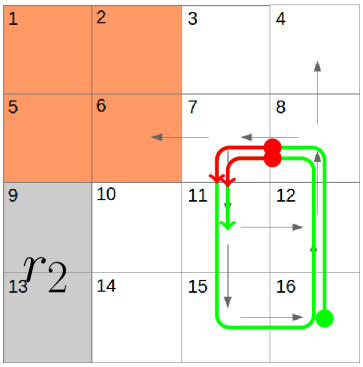


Fig. 5: Map showing configurations for which the revisions ψ_t^e and ψ_t^s from Example 3 apply; a counterstrategy execution trace, as explained in Section 4.2. The green part of the path denotes where *sen* =`False` and the red denotes where *sen* =`True`.

step $\gamma_{\mathcal{X}}^c(q) \cup \gamma_{\mathcal{Y}}^c(q)$. Now, let

$$B'_{nc}(q) = \bigvee_{v \in V_{nc}(q)} \left(\bigwedge_{\pi \in v} \bigcirc \pi \wedge \bigwedge_{\pi \in \mathcal{X}_{nc} \setminus v} \neg \bigcirc \pi \right),$$

be the Boolean representation of $V_{nc}(q)$ at the next execution step.

Our goal is to find, for each $q \in Q_c$, a subset $Q_{cut} \subseteq Q_c$ for which the result

$$B'_{nc}(q) \wedge B_{env}(q) \wedge B_{robot}(q) \wedge \neg WP_{env}|_{\mathcal{X}'_c, \mathcal{Y}'} \neq \texttt{False} \tag{17}$$

is obtained, where WP_{env} is the set of winning positions for the environment and where $(\cdot)|_{\mathcal{X}_c, \mathcal{Y}}$ denotes the *existential abstraction* with respect to propositions in $\mathcal{X}'_c$ and $\mathcal{Y}'$. Consequently, for any $q \in Q_{cut}$ there is valid environment input assignment at the time step after visiting state q that is *not winning* for the environment. One can think of Q_{cut} as being those counterstrategy states where it is possible that the environment has been able to "cut away" a command that will allow the system to proceed to its next goal by applying some environment input. Moreover, for all such environment inputs that are losing for the environment, there exists a valid command the system may take that is winning for the system.

Using Q_{cut}, we formulate a set of liveness assumptions that restrict the environment from *always* behaving in a manner that prevents the system's progress toward its goals. Notice that Q_{cut} contains all states for which there is an environment and system move not in the environment's strategy; however, not all such moves are necessarily winning for the system player. For instance, a state in Q_{cut} could yield an environment input that does not allow the environment player to move strictly closer to its goal yet only allow system moves that place the system further away from its goal.

We therefore form a set $P_{cut} \subseteq Q_{cut}$ for which the system has safe commands that are winning for the system. We use Q_{commit} (from the deadlock counterstrategy) to define the set of states where there exist system moves that lead the system closer to its goals. We populate P_{cut} as follows:

$$\begin{aligned} P_{cut} = \{q \in Q_{cut} \mid \exists v_a \in V_a, \exists q' \in Q_{commit}, \\ \exists q_a \in \delta_a(\gamma^c_{\mathcal{X}}(q), v_a) : \\ \forall \pi \in \mathcal{X}_c, \pi \in \gamma^a_{\mathcal{X}}(q_a) \text{ iff } \pi \in \gamma^c_{\mathcal{X}}(q')\}. \end{aligned} \tag{18}$$

We then apply the environment liveness assumption

$$\square\diamondsuit \bigvee_{q^i \in P_{cut}} \left(B_{robot}(q^i) \wedge \bigwedge_{q^j \in \delta_c(q^i, \gamma^c_{\mathcal{X}}(q^i))} \bigcirc \neg B_{env}(q^j) \right). \tag{19}$$

This liveness formula disallows the environment from denying the system from taking action that lead it closer to its goals, when the system is in a configuration where there is such an action to be taken. Because we are targeting a set of states P_{cut} rather than an entire counterstrategy, the conditions (19) are less restrictive than those in [1]. In that case, let SCC_c be the set of states belonging to a *strongly-connected component* (SCC) in A_c determined using Tarjan's depth-first search algorithm [26]. Then, the revisions are as follows:

$$\square\diamondsuit \bigvee_{q^i \in SCC_c} \left(B_{robot}(q^i) \wedge \bigwedge_{q^j \in \delta_c(q^i, \gamma^c_{\mathcal{X}}(q^i))} \bigcirc \neg B_{env}(q^j) \right). \tag{20}$$

Nonetheless, that approach is adopted as a fallback if the addition of (19) fails to render the specification realizable or if Q_{commit} is empty.

Algorithm 3 uses the counterstrategy from Algorithm 2 to generate liveness assumptions ψ^e_g restricting transitions to cycles of states preventing the system from fulfilling its goals (livelock). Once a candidate liveness assumption is computed, it is checked in Lines 12–19 to ensure that the system's strategy does not contain a sequence of moves that cause the new liveness condition to be falsified. In such cases, realizable returns `False`, and the candidate liveness is removed. The user may elect to accept or discard this formula; if accepted, it is added to the set of runtime certificates.

Example 4 With the specification φ^{abs} in Example 2 along with the deadlock revision (9)–(16), a new counterstrategy is extracted as pictured in Fig. 7 that is free of deadlock. The set of winning states for the environment, shaded orange in Fig. 6a, includes assignments for the environment variable *sen*, also visualized in the figure. From this, we may observe that there are four states, q_3, q_5, q_6, q_8 in Fig. 7 for which there is an alternative assignment to *sen*, namely *sen* =`False`, that does *not* satisfy (17). In each of the other states in the counterstrategy, *sen* =`False` could be replaced with the alternative assignment *sen* =`True`, yet the result will remain in the environment's winning set. Hence, $Q_{cut} = \{q_3, q_5, q_6, q_8\}$.

Algorithm 3 Synthesizing livelock revisions for an realizable specification φ^{mod}.

procedure synthLivelockRevisions(φ^{mod},$\mathcal{A}_c$,WP_{env},$\mathcal{R}$,Q_{commit})
▷ *Eliminate livelocks*
  if $Q_{commit} \neq \emptyset$ **then**
    $Q_{cut} \leftarrow \{q \in Q_c \mid B'_{nc}(q) \wedge B_{env}(q) \wedge B_{robot}(q) \wedge \neg WP_{env}|_{\mathcal{X}'_c,\mathcal{Y}'} \neq \texttt{False}\}$
    $P_{cut} \leftarrow$ Eq. (18)
    $\psi^e_{g,cand}, \psi^e_g \leftarrow$ Eq. (19)
  else
    $\psi^e_{g,cand}, \psi^e_g \leftarrow$ Eq. (20)
  end if
  if $\neg(\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t \wedge \psi^e_{g,cand})$ **then**
    $\psi^e_g \leftarrow \psi^e_g \backslash \psi^e_{g,cand}$
  else
    $\varphi^{try} \leftarrow$ Eq. (2)
    $realiz \leftarrow \mathsf{realizable}(\varphi^{try})$
    if $\neg realiz$ **then**
      $\psi^e_g \leftarrow \psi^e_g \backslash \psi^e_{g,cand}$
      ▷ *System falsifies environment liveness*
    end if
  end if
  ▷ *User feedback*
  for all $R_i \in \mathcal{R}$ **do**
    if then
      print Liveness revisions found for region R_i.
    end if
  end for
  if user accepts livelock revisions **then**
    $\varphi^{mod'} \leftarrow$ Eq. (2)
    $(realiz', \mathcal{A}'_c, WP'_{env}) \leftarrow \mathsf{ctrStrategy}(\varphi^{mod'})$
  end if
  return $realiz'$,$\mathcal{A}'_c$,WP'_{env},$\varphi^{mod'}$
end procedure

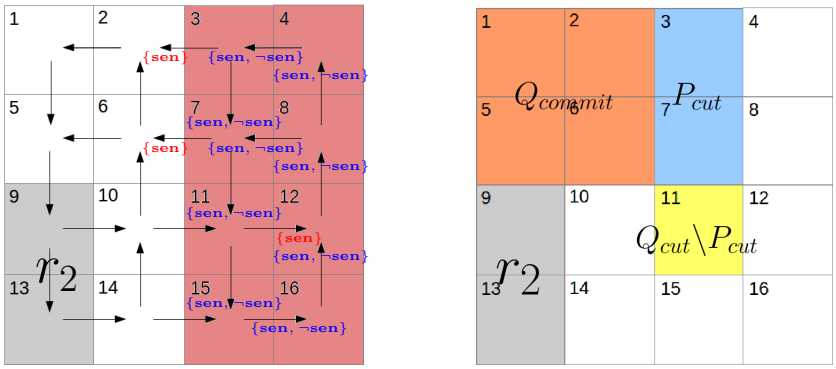
Of these states, q_3, q_6, q_8 are those in which the robot's abstraction has an action (move W) driving the system into configurations matching states within the set Q_{commit} computed in Example 3. The intersection of Q_{cut} and $\{q_3, q_6, q_8\}$ are collected in P_{cut}, corresponding to regions shaded blue in Fig. 6b. Note that this leaves out $\{q_5\}$ (shaded yellow) for which there is an environment move keeping it from immediately realizing an environment goal ($\square\diamondsuit(\neg sen)$) but does not lead the system closer to its goal of reaching r_2.

With P_{cut}, we apply environment liveness revisions ψ^e_g:

$$\square\diamondsuit((x7 \wedge W \wedge \bigcirc \neg sen) \vee (x3 \wedge W \wedge \bigcirc \neg sen) \vee (x7 \wedge S \wedge \bigcirc \neg sen)) . \quad (21)$$

Adding this final revision produces a specification φ^{mod} that is realizable.

Proposition 2 *The alternating application of the approaches in Algorithms 2 and 3 in the context of the flow diagram of Fig. 2 terminates with either a realizable specification or failure.*



(a) (b)

Fig. 6: (a) Partial visualization of the set of winning positions WP_{env}, showing all assignments to the environment proposition *sen* that are in the set WP_{env} in the next step in the execution given the system is currently occupying positions in the red-shaded cells and activating the indicated actions. (b) Map showing regions associated with cut states from Example 4.

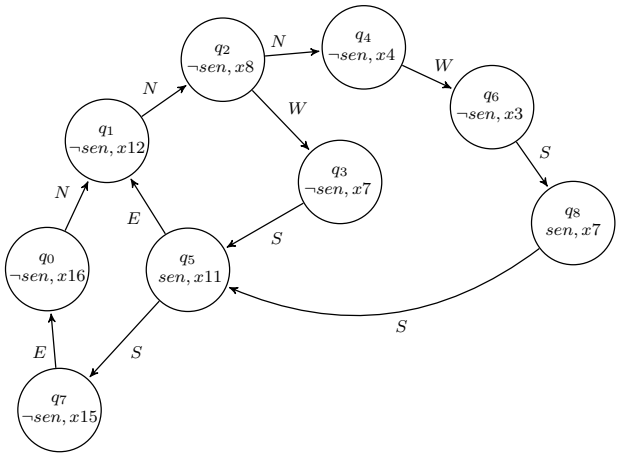


Fig. 7: Deadlock-free counterstrategy for Example 4.

Proof Unrealizability of φ_{abs} with φ realizable implies an inconsistency between the task and the abstraction S_a. The special structure of the problem gives rise to specific selection of the propositions $\mathcal{X}_c \cup \mathcal{Y}$ for B_{robot} and $\mathcal{X}_{nc}$ for B_{env}. Addition of a safety formula (6) or a liveness formula (19) restricts the behavior of $\mathcal{X}_{nc}$ without altering the transitions of S_a. Since the user may preempt the revision process at any step in the algorithm, we assume without loss of generality that the revisions are always accepted.

We prove soundness by showing that, whenever the revisions ψ_g^e and ψ_t^e are found at any iteration $m > 0$, then either $\mathcal{A}_c^m \neq \mathcal{A}_c^{m-1}$ or the revisions falsify the antecedant of $\varphi^{mod,m-1}$, the modified specification φ^{mod} at step $m-1$. The counterstrategy $\mathcal{A}_c^{m-1}$ produces executions σ_c that are either finite (end in deadlock) or infinite (enter an SCC). If $\mathcal{A}_c^{m-1}$ contains deadlock states, then (4) will encode a transition for one such σ_c. If ψ_t^e of (5) does not falsify the antecedant of $\varphi^{mod,m-1}$, then this σ_c will not be an execution accepted by $\mathcal{A}_c^m$. If $\mathcal{A}_c^{m-1}$ contains no deadlock states, it must contain an SCC [1] with an execution σ_c accepted by $\mathcal{A}_c^{m-1}$. Thus, either formula (19) or (20) will produce a ψ_g^e that, if it does not falsify $\varphi^{mod,m-1}$, then at least one execution trace σ_c will not be accepted by $\mathcal{A}_c^m$.

Further iterations of the main loop in Algorithms 2 and 3 that uncover new revisions only adds to the executions σ_c removed and, by application of the induction hypothesis, this results in termination either by recovering a realizable specification, or $\mathcal{A}_c^m = \mathcal{A}_c^{m-1}$ (failure).

Note that the existence of a deadlock or livelock revision is predicated on the reachability of the system goals φ_g^s.

5 Creation of Runtime Certificates

Our feedback to the user represents a certificate that, if upheld, will guarantee the mission under the dynamics. We build this certificate around three types of statements: a *command* given to express the added environment safety revisions, a *consequence* used to describe the outcome of the system safety revisions the system's behaviors, and a *cause* for the revisions stemming from the discrete abstraction. We combine the statements automatically into a template of the form

Because [Cause], then [Command]. If these conditions are upheld, [Consequence].

Given this information, the user may choose to accept either the command or consequence statements depending on their consistency with the original design intent. We describe the process for translating the abstraction and revisions into such statements. We also provide a graphical tool that aids this process by allowing the user to interact with a map of the workspace.

5.1 Parsing the Revisions

By exploiting the iterative aggregation procedure in Section 4.1, we group statements in terms of commit states resulting in certificates that are simpler to parse than would be the case if separate statements were to be given for each region of the configuration space. For each labeled workspace region $R_i \in \mathcal{R}$, we mark those deadlock states (if any exist) from whose predecessors there

exists a command $v_a \in V_a$ (Definition 1) to reach R_i. Those marked as deadlock states are collected in the set $P_{dead}(R_i)$, defined formally as:

$$P_{dead}(R_i) = \{q \in Q_{dead} \mid \forall q' \in \delta^{c^{-1}}(q), \exists v_a \in V_a, \\ \exists q'_a \in \delta_a(\gamma^c_{\mathcal{X}}(q'), v_a) : \forall \pi \in \mathcal{X}_c, \\ \pi \in \gamma^a_{\mathcal{X}}(q'_a) \text{ iff } \pi \in \gamma_a(R_i)\}.$$

In addition to giving the user a graphical representation for the set of the configuration space over which the revisions have been generated (discussed in Sec. 5.2), we verbally provide the user with a conservative metric for this set. In the fixed point computation in Algorithm 1, we keep track of each deadlock state reachable from each state added to Q_{commit}. We use this stored information to find the distances associated with the regions for each state stored in Q_{commit}, and provide the user with a simple numerical metric overapproximating the conditions under which the environment would be required to adhere for the generated revisions to be satisfied. Specifically, this overapproximation is a radius of an enclosing circle projected onto the Cartesian subspace.

Given counterstrategy state $q \in Q_c$, let $[\![\gamma^c_{\mathcal{X}}(q)]\!]$ denote the projection of the region R_i for which $\pi_i = \gamma^c_{\mathcal{X}}(q)$ onto $\mathbb{R}^3 \cap \mathcal{W}$. For each R_i corresponding to a configuration for the system that satisfies some deadlock state in Q_{dead}, we find the relative proximity to a deadlock condition (in terms of physical coordinates) by finding the maximal pairwise distance between any states affected by the deadlock revisions:

$$(q^\star_i, q^\star_{dead,i}) = \underset{\substack{q \in Q_{commit}, \\ q' \in Q_{reach}(q) \cap P_{dead}(R_i)}}{\arg\max} \left| [\![\gamma^c_{\mathcal{X}}(q)]\!] - [\![\gamma^c_{\mathcal{X}}(\delta^{c^{-1}}(q'))]\!] \right|. \tag{22}$$

Here, $|v_{x'}|$ is the Euclidean norm of the real-valued abstraction state $q_a \in Q_a$ represented by a set of propositions $v_{x'} \subseteq \mathcal{X}_c$ that are `True` in that state. The pair of counterstrategy states $q^\star_i$ and $q^\star_{dead,i}$ are those corresponding to a revision for region R_i where the distance is greatest, under the constraint that $q^\star_{dead,i}$ is a deadlock state that is reachable from $q^\star_i \in Q_{commit}$. Note that the distance between the configurations of the two states is:

$$dist_i = \left| [\![\gamma^c_{\mathcal{X}}(q^\star_i)]\!] - [\![\gamma^c_{\mathcal{X}}(\delta^{c^{-1}}(q^\star_{dead,i}))]\!] \right|.$$

5.1.1 Translating Environment Assumptions into Command Statements

Our goal is to provide users with statements such as "Keep sensor *sen* `False` if the robot enters to within N meters of $r2$". We correlate each unique region R_i to the environment proposition assignments prevented by the safety assumption revisions ψ^e_t. Those prevented assignments are given in the formula $\psi_2(q^\star_{dead,i})$. That is, the added environment assumptions prevent the environment from triggering the combination $\psi_2(q^\star_{dead,i})$. The data provided to the user is represented by the triple $(R_i, dist_i, \psi_2(q^\star_{dead,i}))$. The triple can be displayed to the user as follows: "If the robot is within $dist_i$ of region R_i,

then the generated deadlock revisions (for a given counterstrategy) will be satisfied if the environment is not set to $\psi_2(q^\star_{dead,i})$." Note that this metric supplies a sufficient but not necessary condition for satisfying the revisions. That is, there might be executions where the system enters within $dist_i$ with any environment setting yet still be able to satisfy the revision formulas.

5.1.2 Translating System Guarantees into Consequence Statements

In a similar manner to our formation of command statements, we generate consequential statements based on the set of states that are backward-reachable from deadlock states. Such statements are used to convey the added guarantees that are required to recover behavior that is close to that of the topology graph. The triple $(R_i, dist_i, \psi_2(q^\star_{dead,i}))$ yields the statement "If the environment does not set $\psi_2(q^\star_{dead,i})$, then the robot will not move to within $dist_i$ of region R_i." The user can elect to accept or discard these statements (hence controlling whether or not ψ^s_t is inserted into (2)), providing a more refined level of control over the behavior of the system in the presence of the added environment assumptions.

5.1.3 Translating the Abstraction into Causal Statements

Statements are also provided to the user in order to convey to the user the root cause of the added environment assumptions. These are generated as a consequence of the reachability analysis described in Sec. 4.1. Specifically, the pair $(R_i, dist_i)$ produces the statement "If the robot is within $dist_i$ of region R_i, then it cannot avoid ultimately entering R_i.".

The following example illustrates the procedure.

Example 5 In the result of Example 3, let q_{dead} be the deadlock state computed by the counterstrategy corresponding to the configuration $x9$, and designate $Q_{commit} = \{q_1, q_2, q_3, q_4\}$ as the set of commit states for this deadlock. For workspace region r_2, $P_{dead}(r_2) = q_{dead}$, and $Q_{reach}(q_i) = q_{dead}$ for $i = 1, \ldots, 4$. We next determine the pair $(q^\star_2, q^\star_{dead,2})$ to be

$$(q^\star_2, q^\star_{dead,2}) = \quad \arg\max \left\{ \left| \begin{pmatrix} 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right|, \left| \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right|, \right.$$
$$\left. \left| \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right|, \left| \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right| \right\} = (q_2, q_{dead}),$$

where the subscript 2 in the * variables is used to signify the fact that the variables apply to region r_2. Assuming $\eta = 1m$, the corresponding distance is $dist_2 = \sqrt{1^2 + 2^2} = 2.2m$. Finally, reflective of the revisions in (9)–(16), we note the subformula $\psi_2(q^\star_{dead,2}) = sen$.

Therefore, the generated causal statement is: "the robot cannot avoid entering r_2 if it enters to within $2.2m$ of it". LTL formulas in (9)–(12) are summarized as: "if the robot enters to within $2.2m$ of r_2, the environment must

not set the variable *sen* to `True`". Likewise, the LTL formulas in (13)–(16) are summarized as: "if the environment sets *sen* to `True`, the robot will not enter to within $2.2m$ of r_2".

5.2 Graphical Visualization Tool

We aid the user in eliciting the runtime certificates generated above via a graphical user interface (GUI), pictured in Fig. 8, which runs as an extension to the LTLMoP toolkit[2]. The tool allows a user to make queries on different regions and actions with the aid of a map to discover any runtime certificates that have been generated. The generated statements will change depending upon the user's decision to accept or reject the revisions. In the example shown, the region denoted E_dropoff_L indicates an overapproximation and projection of the deadlock-committed states, which has been computed by analysis (e.g. [7]).

In the query example shown in Fig. 8, the user has selected the region E_dropoff_L to query the case where the robot is currently in E_dropoff_L. The user has next selected dropoff_L to indicate that moving to dropoff_L is the commanded action the robot is taking. The certificates for this query is provided in the GUI text box. The GUI aids a user when there are numerous certificates to consider, or when the workspace has been re-partitioned as a consequence of reachability analysis. We examine such a use case in further detail in Sec. 6.

6 Case Studies

In this section, we demonstrate the revisions approach in two example scenarios. The examples serve to demonstrate the effectiveness of our approach when the designer is faced with different specifications and different types of discrete abstractions representing the physics of a mobile robot.

6.1 Abstractions

We begin by defining two special cases of abstraction that fit in the general definition of Definition 1.

6.1.1 Temporally-Grounded Abstractions

The discrete abstraction in a temporal paradigm. By adopting the approach in [24, 21, 32], we may discretize the bounded configuration space $\mathcal{W} \subset \mathbb{R}^n$ and the bounded space of command inputs $\mathcal{U} \subset \mathbb{R}^m$, then define $q'_a \in \delta_a(q_a, v_a)$ to be the set of configurations $\gamma^a_{\mathcal{X}}(q'_a)$ that are reached after some elapsed time τ. Specifically, we denote $[\mathcal{W}]_\eta$ and $[\mathcal{U}]_\mu$ to be, respectively, the uniform grid

[2] https://github.com/VerifiableRobotics/LTLMoP/

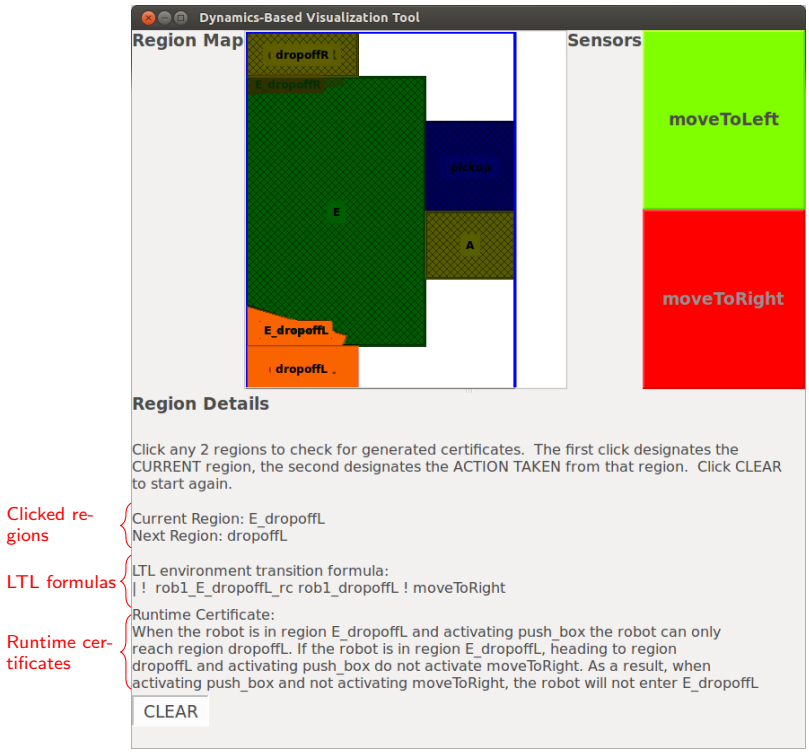


Fig. 8: A screen capture of the certificate visualization tool. The user specifies the current region and action by clicking regions in the workspace map (highlighted orange). Based on this input, the LTL formulas representing the revision matching that selection are displayed in the info box, along with a certificate of the revisions, provided as text-based feedback. Any sensors that are disallowed as per the command statement are painted red in the upper-right list.

on $\mathcal{W}$ discretized with resolution η and $\mathcal{U}$ discretized with resolution μ. This grid is defined as follows:

$$[\mathcal{W}]_\eta := \{x \in \mathcal{W} \mid \exists k \in \mathbb{Z}^n : x = k\eta\}, \tag{23}$$
$$[\mathcal{U}]_\mu := \{u \in \mathcal{U} \mid \exists k \in \mathbb{Z}^m : u = k\mu\}. \tag{24}$$

Note that, in Definition 1, $Q_a = [\mathcal{W}]_\eta$, $V_a = [\mathcal{U}]_\mu$. Also, rather than δ_a being defined as the region under which motion completes, it is defined to complete after some time interval τ. The reader is referred to [5] for full details on the approach to constructing such an abstraction.

6.1.2 Abstractions Grounded on Activation/Completion of Motion

In this case, the transition relation δ_a is defined to be the possible regions $\gamma^a_{\mathcal{X}}(q'_a)$ that may be reached under action $v_a \in V_a$ from region $\gamma^a_{\mathcal{X}}(q_a)$. For a particular transition under δ_a to be defined, the underlying system must guarantee that, under action v_a, the system eventually reaches the set $\gamma^a_{\mathcal{X}}(q'_a)$, $q'_a \in \delta_a(q_a, v_a)$, from any continuous state $\xi \in \gamma^a_{\mathcal{X}}(q_a)$ and remains within the union of $\gamma^a_{\mathcal{X}}(q'_a)$ and $\gamma^a_{\mathcal{X}}(q_a)$. Alternatively, a subset of $\gamma^a_{\mathcal{X}}(q_a)$ may be taken as long as a controller composition property holds (see [7] and references therein for details).

6.2 Revisions in a Finely-Partitioned, Temporal Abstraction

In this case study, we return to the factory scenario in Example 1 using the workspace in Fig. 1. To carry out this task, we select a robot described by a unicycle model that is governed by the kinematic relationship:

$$\dot{x} = v\cos\theta, \qquad \dot{y} = v\sin\theta, \qquad \dot{\theta} = \omega,$$

where the x and y are the Cartesian displacements in meters, θ is the orientation angle, and v and ω are, respectively, the forward and angular velocity inputs to the system. The car model is subjected to the constraint where it may only move with *positive* forward velocity (it cannot stop). An abstraction is generated for the three-dimensional configuration space and two-dimensional input space consisting of 2.2×10^6 states, with the chosen values $\eta = 0.15$, $\mu = 0.2$, $\tau = 0.35$.

We generate a temporally-grounded abstraction, as described in Sec. 6.1.1, using the Pessoa Toolbox.[3] For synthesis, we use the Slugs Synthesis Tool, part of the LTLMoP Toolkit;[4].

The general specification is realizable, producing the controller pictured in Fig. 9; however the specification φ^{abs} (with respect to the unicycle model) is unrealizable. With the approach in Algorithms 2 and 3, we compose revisions that render φ^{mod} realizable. After a counterstrategy is synthesized, revisions are found for a total of 2040 states in the counterstrategy (taking 1020 seconds to synthesize on a laptop PC with a dual-core processor and 8GB memory). A metric for these revisions is generated and the user is prompted with the following:

```
Deadlock revisions found.
When within 1.32 m of station_1, never set environment variable s1_occupied to
True.
Accept? (y/n)
If the environment variable s1_occupied is set to True, then the robot will
never enter to within 1.32 m of station_1.
Accept? (y/n)
```

[3] https://sites.google.com/a/cyphylab.ee.ucla.edu/pessoa/

[4] https://github.com/VerifiableRobotics/slugs

Note that, as our configuration space consists of variables of mixed units, the norm computed in (22) has been projected onto the Cartesian plane. Recall that accepting the first statement (environment behaviors) removes deadlocks associated with entry into station_1, while accepting the revisions to the second statement does not alter the realizability of the resulting specification, but instead produces different system behaviors in proximity to station_1.

A second prompt is given:

```
When within 1.44 meters of station_2, never set environment variable s2_occupied
to True.
Accept? (y/n)
If the environment variable s2_occupied is set to True, then the robot will
never enter to within 1.44 m of station_2.
Accept? (y/n)
```

At this point, should the user accept both revisions, a new counterstrategy is synthesized containing no deadlock states. The user is prompted again:

```
Livelock revisions found. When within 1.35 meters of station_1, always eventually
set environment variable s1_occupied to False.
Accept? (y/n)
Livelock revisions found. When within 1.46 meters of station_2, always eventually
set environment variable s2_occupied to False.
Accept? (y/n)
```

Note that the revisions are associated with a set of counterstrategy states in P_{cut}. The locative commands are computed in a similar manner to the deadlock revisions by replacing the states within P_{cut} in place of Q_{commit}.

This time, the specification is realizable if the user accepts this revision. The resulting execution for the controller is as shown in Fig. 10. The trajectories pictured in the figure represent evolutions of the continuous nonlinear system when commanded by the synthesized controller. Forward integration is applied to solve the equations of motion using an integration step size of 0.001 sec. Note that the system in the figure infinitely often visits the three regions and is able to react to a change in the environment. In Fig. 10a, the system avoids the region station_1 whenever s1_occupied turns True, this happens at distances greater than 1.32 m of station_1. A similar result is seen in Fig. 10b. These behaviors are consistent with the intended behaviors encoded by the specification in Example 1.

6.3 Workspace Re-Partitioning in the Activation/Completion Paradigm

In this case study, we examine the use of our approach to fulfill a specification where the robot is tasked with moving packages from a pick-up area to one of two drop-off locations, as pictured in Fig. 11. The specification is, "Visit the loading area. If push_box is active and go_to_left is requested, visit dropoff_L. If push_box is active and go_to_right is requested, visit dropoff_R. Activate push_box when in pickup and deactivate push_box when in dropoff_L or dropoff_R."

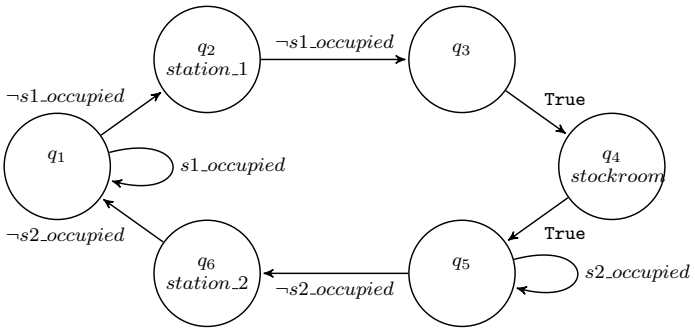
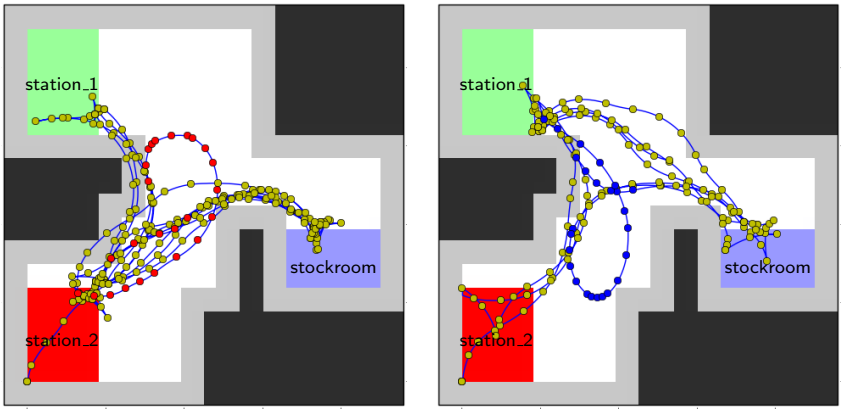


Fig. 9: Controller for φ in Example 1. Edges are labeled with the disjunction of assignments in $\mathcal{X}$ that may be assumed for that transition.



(a) (b)

Fig. 10: Continuous trajectories for the nonlinear unicycle abstraction in a 5×5 workspace, where the robot is initialized at the lower-left corner of the workspace. Dots along the trajectory indicate the position of the robot when a new control command is received (a time step of 0.35 seconds). Color indicates the state of the environment (red: s1_occupied; blue: s2_occupied). (a) shows a trajectory when the s1_occupied sensor is activated. (b) shows a trajectory when the s2_occupied sensor is activated.

We employ a KUKA youBot to perform the task, which operates on an omnidirectional base whose position and orientation is measured in real time. Packages are moved by way of pushing them along the ground using the robot's front fender. There is one action (treated as a system variable) in this scenario, push_box, which is `True` whenever the robot is moving a package and `False` otherwise.

The discrete abstraction for the robot is created using the activation/completion approach of Sec. 6.1.2. When pushing the box, we impose conditions

of the under-actuated unicycle model, constrained with fixed forward velocity, as explained in Sec. 6.2 in order to assure that the box always maintains contact with the robot. When the robot is no longer required to push the box or when it must disengage with the box, holonomic (fully-actuated) dynamics are imposed to allow the robot to move freely.

We synthesize, using the procedure in [7], a finite-state machine satisfying the mission specification consisting of 6 states and 9 transitions. Controllers were synthesized for the dynamical system using a low-level controller synthesis procedure described in [7] that uses the FSM resulting from synthesis of the general formulas to compute controllers that respect reachability under the dynamics, guaranteeing any sequence of FSM states from any initial robot state selected within the reachable set. If the process fails to compute a controller for some behavior in the FSM, the workspace is re-partitioned and a change to the discrete abstraction is triggered. If the resulting specification is unrealizable, this prompts a call to the revisions approach discussed in this paper to uncover any certificates associated with an unrealizable specification.

In this case study, the original specification could not be implemented using the imposed dynamics, necessitating an update to the abstraction and the creation of new regions $R_{middle,L}$ and $R_{middle,R}$ based on reachability computations ($R_{middle,L}$ is as indicated in Fig. 11). Using the proposed revisions approach and calls to the slugs synthesis tool, we automatically generate runtime certificates that restrict the environment's behavior and alter the system's behavior to accommodate the robot's limited capability for movement in these regions. In Fig. 8, one such certificate is shown for the case where the robot is in $R_{middle,L}$ and activating motion to dropoff_L. The configuration under consideration are highlighted in orange, and the sensor value (request) that is required to be inactive for that configuration is highlighted in red. A similar set of certificates was generated for $R_{middle,R}$.

The execution of the controllers generated as a result of the generated revisions are shown in Fig. 11. As the robot is heading to dropoff_L but while still outside $R_{middle,L}$, the environment (human operator) is not violating the certificate if the request is changed from go_to_left to go_to_right, and hence the system is guaranteed to react to the environment and execute the task.

7 Conclusions

In this paper, we have described an automatic approach for generating runtime certificates for missions carried out on physical systems with dynamics. Our contribution is an approach that makes use of the problem structure for reactive missions to arrive at certificates that preserve the behavior of the physical system when executing a controller generated from a general specification that is agnostic to the dynamics. The proposed approach features a mechanism for providing feedback to the user as text-based feedback, aided by a GUI, and enables the freedom to accept or reject any such proposed formula at synthesis time. A key benefit of our framework is the ability to generate a

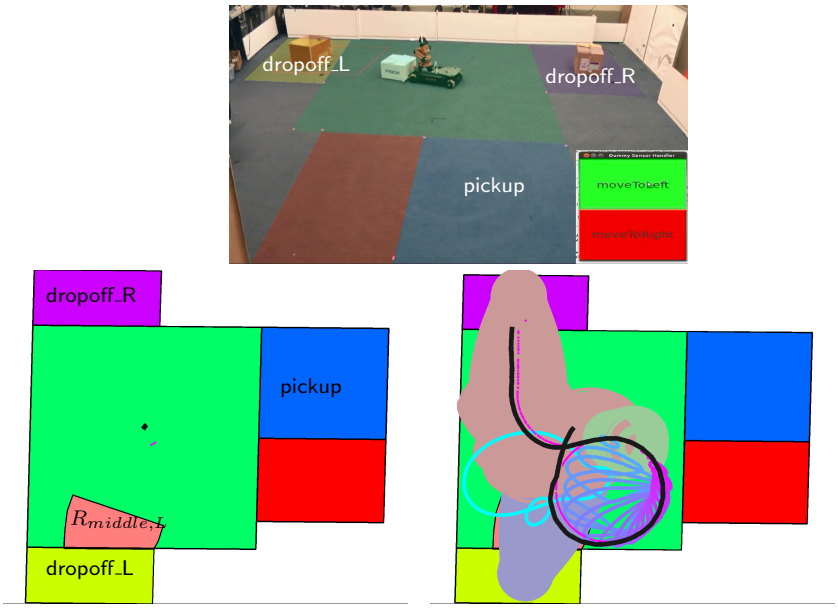


Fig. 11: Problem set up for the box-transportation scenario. The top image shows the KUKA youBot performing the task of delivering a box to the appropriate region as determined by the sensor. The map is displayed at the bottom left. The new region as a result of the reachability-based re-partitioning, $R_{middle,L}$, is shown in pink. The robot's trajectory is shown in black at the bottom right, along with the reachable sets for the activated controllers, and a nominal trajectory (magenta). A runtime certificate is generated (indicated in Fig. 8) that indicates that the sensor should not change go_to_left to go_to_right when the robot is in $R_{middle,L}$. For full details, the reader is referred to [7].

small number of revisions for the task, and those that are generated are concise enough to be easily interpreted. This provides the user the best opportunity at synthesizing a controller that is consistent with the original design intent of the specification.

Future work includes providing a means for suggesting a richer set of possible revisions to give as feedback to the user, thereby offering him or her a multiplicity of possible options to apply (e.g. trading off modifying the system's behavior vs. restricting the environment). Such an extension will involve mining more complex formulas from the synthesis game and automatically translating such formulas into easy-to-understand explanations. Future efforts toward user studies would give the ability to objectively evaluate the effectiveness of the tool as users create their own specifications for (possibly complex) robotic systems.

Acknowledgements The authors thank Paulo Tabuada for insightful discussions and assistance with PESSOA, Vasumathi Raman and Salar Moarref for insightful discussions relating to synthesis of counterstrategy-based environment revisions, and Divyansha Sehgal for her assistance with the visualization tool. The authors lastly thank the anonymous reviewers for their constructive critique.

References

1. Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of gr(1) temporal logic specifications. In: Formal Methods in Computer-Aided Design (FMCAD 2013), pp. 26–33 (2013)
2. Bhatia, A., Kavraki, L., Vardi, M.: Sampling-based motion planning with temporal goals. In: IEEE International Conference on Robotics and Automation (ICRA 2010), pp. 2689–2696. IEEE (2010)
3. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – a new requirements analysis tool with synthesis, pp. 425–429. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-14295-6_37. URL `http://dx.doi.org/10.1007/978-3-642-14295-6_37`
4. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012)
5. DeCastro, J.A., Ehlers, R., Rungger, M., Balkan, A., Tabuada, P., Kress-Gazit, H.: Dynamics-based reactive synthesis and automated revisions for high-level robot control. CoRR **abs/1410.6375** (2014). URL `http://arxiv.org/abs/1410.6375`
6. DeCastro, J.A., Kress-Gazit, H.: Synthesis of nonlinear continuous controllers for verifiably-correct high-level, reactive behaviors. The International Journal of Robotics Research **34**(3), 378–394 (2015). DOI 10.1177/0278364914557736. URL `http://ijr.sagepub.com/content/34/3/378`
7. DeCastro, J.A., Kress-Gazit, H.: Nonlinear controller synthesis and automatic workspace partitioning for reactive high-level behaviors. In: Proceedings of the 19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC). Vienna, Austria (2016)
8. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, pp. 333–339 (2016). DOI 10.1007/978-3-319-41540-6_18. URL `http://dx.doi.org/10.1007/978-3-319-41540-6_18`
9. Fainekos, G.E.: Revising temporal logic specifications for motion planning. In: Proceedings of the IEEE Conference on Robotics and Automation (2011)
10. Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic robots. Automatica **45**(2), 343 – 352 (2009). DOI DOI: 10.1016/j.automatica.2008.08.008
11. Girard, A., Pola, G., Tabuada, P.: Approximately bisimilar symbolic models for incrementally stable switched systems. Automatic Control, IEEE Transactions on **55**(1), 116–126 (2010)
12. Kloetzer, M., Belta, C.: Dealing with nondeterminism in symbolic control. In: M. Egerstedt, B. Mishra (eds.) Hybrid Systems: Computation and Control, 11th International Workshop (HSCC 2008), *Lecture Notes in Computer Science*, vol. 4981, pp. 287–300. Springer (2008)
13. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications using simple counterstrategies. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, pp. 152–159 (2009)
14. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal logic based reactive mission and motion planning. IEEE Transactions on Robotics **25**(6), 1370–1381 (2009)
15. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, pp. 43–50 (2011)

16. Li, W., Sadigh, D., Sastry, S.S., Seshia, S.A.: Synthesis for human-in-the-loop control systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, pp. 470–484 (2014)
17. Liu, J., Ozay, N.: Abstraction, discretization, and robustness in temporal logic control of dynamical systems. In: Proc. of the 17th Int. Conf. on Hybrid Systems: Computation and Control (HSCC'14), pp. 293–302 (2014). DOI 10.1145/2562059.2562137
18. Liu, J., Ozay, N., Topcu, U., Murray, R.M.: Synthesis of reactive switching protocols from temporal logic specifications. IEEE Trans. Automat. Contr. **58**(7), 1771–1785 (2013)
19. Maly, M., Lahijanian, M., Kavraki, L.E., Kress-Gazit, H., Vardi, M.Y.: Iterative temporal motion planning for hybrid systems in partially unknown environments. In: ACM International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 353–362. ACM, ACM, Philadelphia, PA, USA (2013)
20. Nilsson, P., Ozay, N.: Incremental synthesis of switching protocols via abstraction refinement. In: 53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014, pp. 6246–6253. IEEE (2014). DOI 10.1109/CDC.2014.7040368. URL `http://dx.doi.org/10.1109/CDC.2014.7040368`
21. Pola, G., Girard, A., Tabuada, P.: Approximately bisimilar symbolic models for nonlinear control systems. Automatica **44**(10), 2508–2516 (2008)
22. Raman, V., Kress-Gazit, H.: Towards minimal explanations of unsynthesizability for high-level robot behaviors. In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2013) (2013)
23. Raman, V., Piterman, N., Kress-Gazit, H.: Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In: IEEE International Conference on Robotics and Automation, pp. 4075–4081. Karlsruhe, Germany (2013)
24. Reißig, G.: Computing abstractions of nonlinear systems. IEEE Transactions on Automatic Control **56**(11), 2583–2598 (2011)
25. Tabuada, P., Pappas, G.J.: Linear time logic control of discrete-time linear systems. IEEE Trans. Automat. Contr. **51**(12), 1862–1877 (2006)
26. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972). DOI 10.1137/0201010. URL `http://dx.doi.org/10.1137/0201010`
27. Tumova, J., Yordanov, B., Belta, C., Cerna, I., Barnat, J.: A symbolic approach to controlling piecewise affine systems. In: 49th IEEE Conference on Decision and Control (CDC), pp. 4230 –4235 (2010). DOI 10.1109/CDC.2010.5717316. URL `pdf/cdc10b.pdf`
28. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for concurrency, pp. 238–266. Springer (1996)
29. Wolff, E.M., Topcu, U., Murray, R.M.: Automaton-guided controller synthesis for nonlinear systems with temporal logic. In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2013) (2013)
30. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon control for temporal logic specifications. In: Proc. of the 13th Int. Conf. on Hybrid Systems: Computation and Control (HSCC'10) (2010)
31. Yordanov, B., Tumova, J., Cerna, I., Barnat, J., Belta, C.: Temporal logic control of discrete-time piecewise affine systems. Automatic Control, IEEE Transactions on **57**(6), 1491–1504 (2012)
32. Zamani, M., Pola, G., Mazo, M., Tabuada, P.: Symbolic models for nonlinear control systems without stability assumptions. IEEE Transactions on Automatic Control **57**(7), 1804–1809 (2012)