EXPLAINING UNSYNTHESIZABILITY OF HIGH-LEVEL ROBOT BEHAVIORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Vasumathi Raman

August 2013

EXPLAINING UNSYNTHESIZABILITY OF HIGH-LEVEL ROBOT BEHAVIORS

Vasumathi Raman, Ph.D.

Cornell University 2013

As robots become increasingly capable and general-purpose, it is desirable for them to be easily controllable by a wide variety of users. The near future will likely see robots in homes and offices, performing everyday tasks such as fetching coffee and tidying rooms. There is therefore a growing need for non-expert users to be able to easily program robots performing complex high-level tasks. Such high-level tasks include behaviors comprising non-trivial sequences of actions, reacting to external events, and achieving repeated goals.

Recent advances in the application of formal methods to robot control have enabled automated synthesis of correct-by-construction hybrid controllers for complex high-level tasks. These approaches use a discrete abstraction of the robot workspace and a temporal logic specification of the environment assumptions and desired robot behavior, and yield controllers that are provably correct with respect to this abstraction and specification. However, there are many remaining challenges in ensuring that a user-defined specification yields a robot controller that achieves the desired high-level behavior. This dissertation addresses several causes of failure resulting from logical implications of the specification itself, as well as those arising because of inconsistencies between the discrete abstraction and the continuous execution domain.

Work on three main challenges is described. The first is an *algorithm for the analysis of logic specifications*, which provides a high-level cause of failure for specifications that have no implementation, or *unsynthesizable* specifications. An *interactive game* is also introduced, allowing users to explore the cause of unsynthesizability. The second is the

identification of a *minimal* explanation of failure: several techniques are presented to identify a *core* subset of the specification that causes unsynthesizability.

The third problem addressed is the definition of an appropriate *timing semantics for abstraction and execution* of hybrid controllers synthesized from high-level specifications. Several controller-synthesis frameworks are compared, and their suitability to different problem domains discussed, based on their underlying assumptions and properties of the resulting continuous behaviors.

BIOGRAPHICAL SKETCH

Vasumathi Raman was born in Mumbai, India in 1986. She completed her primary and secondary schooling in Bombay, before earning an International Baccalaureate Diploma at the Mahindra United World College of India from 2001 to 2003. She then obtained a Bachelor of Arts in Computer Science and Mathematics from Wellesley College, Massachusetts, where she studied from 2003 to 2007. In August 2007, she entered the Department of Computer Science at Cornell University to pursue her doctorate, which she received in 2013.

for Ma

ACKNOWLEDGEMENTS

My first thanks are to my advisor, Hadas Kress-Gazit, for her support and encouragement. I am indebted to her for giving me the opportunity to dive head-first into robotics, a field I care deeply about but had next to no formal experience with. Her confidence in me is a source of great personal pride, as are the contributions I have been able to make on the topics described in this dissertation.

I thank Joe Halpern and Rafael Pass, with whom I greatly enjoyed working during my first three years at Cornell. In particular, I thank Joe for his excellent advice and support through my many moments of discouragement during the roller-coaster ride that is the PhD. I would also like to thank the rest of my graduate committee, including Bart Selman for his advice on the use of SAT solvers, and David Easley for guiding my graduate minor in Economics.

I have had the pleasure of collaborating with an exceptional set of coauthors on the publications that culminated in this dissertation. I thank Cameron Finucane for always being willing to discuss a crazy new idea, and for implementing the interactive game for unrealizable specifications in Chapter 4. I am also grateful to Nir Piterman for suggesting a timing semantics for robot controllers with arbitrary execution durations for Chapter 6.

During my time in the Verifiable Robotics Group at Cornell, I have had the opportunity to work with many wonderful students and staff. Thanks to Ben Johnson for always being willing to provide constructive criticism on a paper draft or a nascent idea, and to Rüdiger Ehlers for being a seemingly never-ending source of formal methods knowledge. I also thank Jim Jing for indulging my frequent requests for cool photographs of robots doing silly things.

Thanks to all the faculty and students in the Cornell Computer Science Department for making this experience fun, above all else. Thanks also to Becky Stewart

and Stephanie Meik for the many years of cheerful and efficient administrative support.

My deep gratitude to Eoin for his constant encouragement, and for bringing much-needed cheer, calm and sanity to my life these past few months. Thanks also to Brian and Fiona for accommodating my need for a working vacation with such grace. Most of all, I thank my parents, Kannamma and Sundararaman, for their unconditional love and unwavering belief in my abilities – I cannot believe they trusted me enough to let me move across the globe at seventeen.

The research in this dissertation was supported by NSF CAREER CNS-0953365, ARO MURI (SUBTLE) W911NF-07-1-0216, NSF ExCAPE and DARPA N66001-12-1-4250.

I thank all future readers of this dissertation. Thank you for reading, please enjoy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As robots become more ubiquitous, multi-capable and general-purpose, it is desirable for them to be easily controllable by non-expert users. The near future will likely see robots in homes and offices, performing everyday tasks such as fetching coffee and tidying rooms. The main context for this is provided by the growing need for non-expert users to be able to easily program robots performing complex high-level tasks. Such high-level tasks include behaviors comprising non-trivial sequences of actions, potentially including reacting to external events, and repeated goals; other examples include search and rescue missions and the DARPA Urban Challenge [36].

The challenge of programming robots to perform these tasks has until recently been the domain of experts, and typically involved hard-coded high-level implementations and ad-hoc use of low-level techniques such as path-planning during execution. However, with such approaches, it is often not known a priori whether the proposed implementation actually captures the high-level requirements, or whether the intended behavior is even achievable. This motivates the application of formal methods to guarantee that the implemented plans will produce the desired behavior. Recent advances in the application of formal methods to robot control have enabled automated synthesis of correct-by-construction hybrid controllers for complex high-level tasks (e.g., [35, 41, 7, 34, 26, 43]). In particular, temporal logic synthesis has been successfully applied to automatically synthesize autonomous robot controllers [26, 43]. Synthesis-based approaches use a discrete abstraction of the robot workspace and a temporal logic specification of the environment assumptions and desired robot behavior. The robot behavior specification includes safety and liveness requirements, and initial conditions for both robot and environment. The term *environment* in this context includes the physical

workspace as well as external events captured using the robot's sensors, including other robots. The robot controllers generated represent a rich set of infinite behaviors, and are provably correct-by-construction: the closed loop system they form is guaranteed to satisfy the desired specification in any admissible environment (i.e. any environment that satisfies the modeled assumptions).

However, there are several challenges involved in ensuring that a user-defined specification yields a controller that achieves the desired behavior. In the above formal approaches, when the specification is feasible, a controller is generated; however, when there exist admissible environments in which the robot fails to achieve the desired behavior, controller synthesis fails – such a specification is called *unsynthesizable*. An unsynthesizable specification is either *unsatisfiable*, in which case the robot cannot achieve the desired behavior no matter what happens in the environment (e.g. if the task requires patrolling a disconnected workspace), or *unrealizable*, in which case there exists at least one environment that can thwart the robot. For example, if the environment can disconnect an otherwise connected workspace, such as by closing a door, a specification requiring the robot to patrol the workspace is merely unrealizable rather than unsatisfiable. Note that in the formal methods literature, the term unrealizable is usually used to denote both unsatisfiable and unrealizable (since unsatisfiability is just a special case of unrealizability), but in this work we distinguish the two. In addition, a specification is unsynthesizable if the environment can force the system into either a safety violation (termed *deadlock*) or a liveness violation (*livelock*).

When the specification is unsynthesizable (there exists no implementing controller), synthesis-based approaches fail to produce the desired controller, but do not typically provide a source of failure. Moreover, even when synthesis is possible, the generated controller (which fulfills the specification) may produce undesirable or trivial behavior,

such as a vacuous controller consisting of a single state with no transitions, for reasons involving unsatisfiability or unrealizability of the environment assumptions and the syntactic structure of the class of specifications considered. This can make troubleshooting specifications an ad hoc and unstructured process.

For example, consider the specification in Listing 1, intended to produce a controller for the hide-and-seek behavior described in Example 1 in Chapter 3. Sentences in a structured language [11, 25] describe the desired robot behavior and assumptions on the environment. However, this specification is unsynthesizable, and there exists no controller to implement the desired behavior in every admissible environment, i.e. every environment that fulfills the specified assumptions; the reason for unsynthesizability is not obvious without further analysis.

The first two chapters of this dissertation address the problem of analyzing unsynthesizable specifications and providing the user with feedback on them. Chapter 4 presents an algorithm for providing initial feedback on a specification; the granularity of the feedback provided in this chapter corresponds to the structure of the specification. Feedback is presented in the form of highlighted sentences in the structured English specification. In addition, a domain-specific interface is introduced to present the cause of failure to the user in the form of an interactive game, which allows the user to take on the role of the system, and see how the environment is able to prevent the desired robot behavior. Chapter 5 presents techniques for further analysis, enabling more specific feedback on an unsynthesizable specification in the form of a *minimal unsynthesizable core*.

Chapter 6 addresses a different challenge that arises in the automatic construction of hybrid controllers for robot systems. In the synthesis-based approach to robot control, an implementing automaton obtained via LTL synthesis is viewed as a hybrid controller, with each discrete transition implemented by executing the relevant low-level

controllers to move between discrete states. Consequently, a single transition between discrete states may correspond to the execution of several low-level controllers. In general, a robot with multiple action capabilities will use low-level controllers that take varying amounts of time to complete. When reasoning about correctness of continuous execution, most approaches make assumptions about the physical execution of actions given a discrete implementation, such as when actions will complete relative to each other, and possible changes in the robot's environment while it is performing various actions. Relaxing these assumptions gives rise to a number of challenges in the continuous implementation of automatically-synthesized hybrid controllers.

Consider a humanoid Aldebaran Nao robot whose actions include waving and walking; walking between regions of interest takes significantly longer than waving. Suppose the Nao is in one region, and is instructed to go to a different room and to wave. If the two actions are activated simultaneously, then the Nao will wave in both rooms, since turning on the waving action takes less time to complete than the motion between rooms. In general, since different actions take varying amounts of time to complete, their continuous execution may cause the robot system to pass through continuous states that correspond to several distinct intermediate discrete states, some of which may contradict the system specification. In the example above, if the robot safety conditions disallow the action of waving in the original room, then the intermediate state that occurs while the robot is moving between rooms while waving is unsafe. However, the automaton synthesized on the discrete problem abstraction will not recognize this as an invalid transition: discrete transitions are assumed to execute instantaneously, so the existence of these intermediate states is not modeled.

The above observation motivates a controller synthesis framework that ensures safety of continuous execution for every discrete transition. On the other hand, if no

such safe controller exists (but the specification is realizable, and a controller with unsafe continuous transitions does exist), we wish to alert the user to this problem with the continuous implementation of the specification. This problem is unique to the robotics domain, and does not usually present itself in other formal methods applications like circuit design. Chapter 6 presents several approaches to discrete synthesis and continuous execution, and compares the assumptions they make on the robot's physical capabilities and the environment in which it operates.

The above projects are implemented within the Linear Temporal Logic MissiOn Planning (LTLMoP) toolkit[11, 47]. LTLMoP is an open source, modular, Python-based toolkit that allows a user to input structured English specifications describing high-level robot behavior, and automatically generates and implements the relevant hybrid controllers using the approach of [26]. The synthesized controllers can be embedded within a simulator or used with physical robots. The most recent version of LTLMoP can be downloaded online[1].

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

CHAPTER 2

RELATED WORK

The application of formal methods to robotics is a new but growing field. Provably robust solutions to point-to-point navigation problems (e.g., "move from position A to B via C") have been extensively explored; these include the use of potential functions, Voronoi diagrams, cell decompositions and probabilistic road maps [28, 33]. Recently, a number of frameworks have been proposed for the verifiable integration of high-level planning with continuous control. Most rely on an abstraction of the underlying system as a discrete transition system, and use model checking [19] and other formal techniques to synthesize control laws (e.g. [35, 41, 7, 34]) on this discrete model. The desired properties are usually expressed using some flavor of temporal logic, such as Linear Temporal Logic (LTL)[4], which allows synthesis of hybrid controllers under several frameworks, and is expressive enough for specifications that include the desired reactive constraints and sequencing of goals. Some works apply efficient synthesis techniques such as [37] to automatically generate provably correct, closed loop, low-level robot controllers that satisfy high-level reactive behaviors specified as LTL formulas [26, 43]. Specifications describe the robot's goals and assumptions on the environment it operates in, using a discrete abstraction.

The problem of explaining robot behaviors that cannot be achieved has only recently been addressed [47, 48, 38, 23, 31, 30]. The authors in [23, 31, 30] address the problem of revising specifications that are not satisfied on a given robot system. In [23], the authors propose a method for revising unsatisfiable LTL specifications. They define a partial order on LTL formulas, and the notion of a valid relaxation for an LTL specification, which informally corresponds to the set of formulas "greater than" formulas in the specification according to this partial order. Formula relaxation for unreachable

states is accomplished by recursively removing all positive occurrences of unreachable propositions in a manner similar to the fixpoint calculation described in Section 4.3. Specifications with logical inconsistencies are revised by augmenting the synchronous product of the robot and environment specifications with previously disallowed transitions as needed to achieve the goal state. In [31, 30], the same authors present exact and approximate algorithms for finding minimal revisions of specification automata, by removing the minimum number of constraints from the unsatisfiable specification. They too encode the revision problem as an instance of Boolean satisability, and solve it using efficient SAT solvers; this is similar to the approach presented in Chapter 5. However, the work presented in this dissertation differs in its objective, which is to provide feedback on existing specifications, not rewrite them. Moreover, this work deals with reactive specifications whereas [23, 31, 30] consider non-reactive plans.

Although explaining unachievable behaviors has only recently been studied in the context of robotics, there has been considerable prior work on unsatisfiability and unrealizability of LTL in the formal methods literature, and the problem of identifying small causes of failure has been studied from several perspectives. For unsatisfiable LTL formulas, the authors of [51] suggest a number of notions of unsatisfiable cores, tied to the corresponding method of extraction. These include definitions based on the syntactic structure of the formula parse tree, subsets of conjuncts in various conjunctive normal forms, resolution proofs from bounded model-checking (BMC), and tableaux constructions. The authors of [9] use the formal definition of causality of [29] to explain counterexamples provided by model-checkers on unsatisfiable LTL formulas; the advantage of this method is the flexibility of defining an appropriate causal model.

The technique of extracting an unsatisfiable core from a BMC resolution proof is one that is well-used in the Boolean satisfiability (SAT) and SAT Modulo Theories (SMT)

(e.g., [24, 13]) literature. A similar technique was used in [42] for debugging declarative specifications. In that work, the abstract syntax tree (AST) of an inconsistent specification was translated to CNF, an unsatisfiable core was extracted from the CNF, and the result was mapped back to the relevant parts of the AST. The approach in [42] only generalizes to specification languages that are reducible to SAT, a set which does not include LTL; Chapter 5 presents a similar approach, using SAT solvers to identify unsatisfiable cores for LTL.

The authors of [2] also attempted to generalize the idea of unsatisfiable cores to the case of temporal logic using SAT-based bounded model checkers; their approach is very similar to that used in this paper for the case of unsatisfiability. In [2], temporal atoms of the original LTL specification were associated with activation variables, which were then used to augment the formulas used by a SAT-based bounded model checker. The result, in the case of an unsatisfiable LTL formula, was a subset of the activation variables corresponding to the atoms that cannot be satisfied simultaneously. This is similar to the approach presented in Chapter 5 for identifying unsatisfiable cores, in that the SAT formulas used to determine the core are exactly those that would be used by a bounded model checker. However, a major difference is that this work does not use activation variables in order to identify conjuncts in the core, but maintains a mapping from the original formula to clauses in the SAT instance.

In the context of unrealizability, the authors of [1] propose definitions for helpful assumptions and guarantees, and compute minimal explanations of unrealizability (i.e., unrealizable cores) by iteratively expelling unhelpful constraints. Their algorithm assumes an external realizability checker, which is treated as a black box, and performs iterated realizability tests. This work will draw on the same iterative realizability testing techniques in Section 5.4. The authors in [32] use model-based diagnosis to remove not

only guarantees but also irrelevant output signals from the specification. These output signals are those that can be set arbitrarily without affecting the unrealizability of the specification. Model-based diagnoses provide more information than a single unrealizable core, but requires the computation of many unrealizable cores. In [32]. this is accomplished using techniques similar to those in [1], which in turn require many realizability checks. The main relative advantage of the work presented in this dissertation is that it reduces the number of realizability checks required for most specifications, as detailed in Sections 5.2 and 5.3.

To identify and eliminate the source of unrealizability, some works like [52, 12] provide a minimal set of additional environment assumptions that, if added, would make the specification realizable; this is accomplished in [12] using efficient analysis of turn-based probabilistic games, and in [52] by mining the environment counterstrategy. On the other hand, the work presented in this dissertation takes the environment assumptions as fixed, and the goal is to compute a minimal subset of the robot guarantees that is unrealizable. Seen from another perspective, this work presumes that the assumptions accurately capture the specification designer's understanding of the robot's environment, and provides the source of failure in the specified guarantees.

The research presented in this dissertation is among the first to analyze high-level specifications in the robotics domain. The techniques applied in Chapter 4 are closely related to those in [39], whose authors implement a set of sophisticated specification analyses in an interactive tool [40] for debugging hardware design specifications. In contrast to this previous work, which presents visual information to the user in the form of binary signals, the interactive game presented in this work is better adapted to the robot domain, as described in Sections 4.4 and 5.2.3.

This is also one of the first works to consider the safety and correctness of continuous executions of synthesized controllers arising from the physical nature of the problem domain. There are a few previous works that incorporate the continuous nature of the physical execution during the discrete synthesis process. For example, the authors of [22, 20] evaluate discrete controllers on optimality with respect to a continuous metric based on the physical workspace, and extract more optimal solutions at synthesis time. The problem of synthesizing provably correct continuous control has however only recently been addressed [45, 50]. Chapter 6 reviews those works, and compares them. Further, it includes details of the modified synthesis algorithm that enables efficient synthesis for the approach in [50].

CHAPTER 3

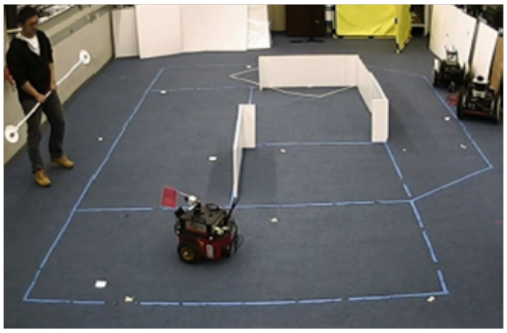
BACKGROUND

The tasks considered in this work involve a robot operating in a known workspace, whose behavior (motion and actions) depends on information gathered at runtime from its sensors about events in the environment. Tasks may also include infinitely repeated behaviors such as patrolling a set of locations.

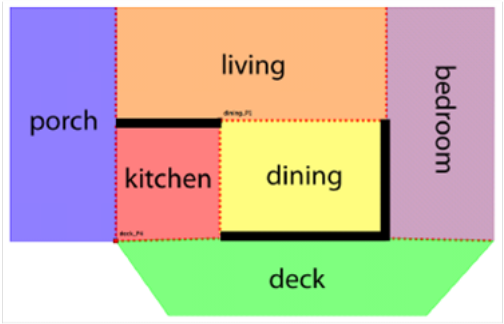
Example 1. Consider the construction of a controller for a robot playing hide and seek in the workspace depicted in Figure 3.1. The robot starts by counting while the other player hides. When it hears the ready whistle, it takes on the role of seeker and looks for the other player. When it has found the other player, it takes on the role of hiding. Once it has been found, it reverts to counting and repeats the cycle.

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

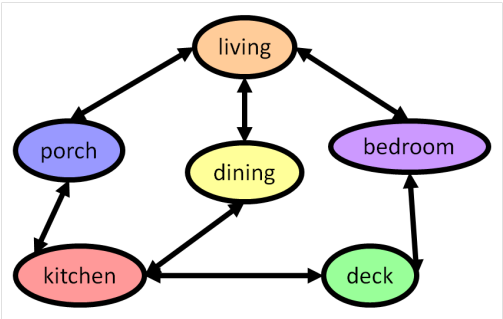
Consider the specification in Listing 1, intended to produce a controller for the above behavior. Sentences in a structured language [11] describe the desired robot behavior



(a) Real Workspace



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



(c) Adjacency Graph

Figure 3.1: Workspace Abstraction and Representation [48]

and assumptions on the environment.

Given the inherent continuous nature of the robotics domain, applying formal methods to the construction of high-level robot controllers requires a discrete abstraction of the problem to enable description with a formal language. Details on the discrete abstraction used in this work can be found in [26]. The formal language used for high-level specifications in this work is Linear Temporal Logic (LTL) [4].

Listing 1 Example of unsynthesizable hide-and-seek specification

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

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

# Mutual exclusion between roles
Always (not seeking or not hiding)
Always (not hiding or not counting)
Always (not counting or not seeking)

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

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

3.1 Linear Temporal Logic

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

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

where $\neg$ is negation, $\vee$ is disjunction, $\bigcirc$ is "next" , and $\mathcal{U}$ is "until". Boolean constants `True` and `False` are defined as usual: $\texttt{True} = \pi \vee \neg\pi$ and $\texttt{False} = \neg\texttt{True}$. Conjunction ($\wedge$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), "eventually" ($\Diamond \varphi = \texttt{True}\mathcal{U}\varphi$) and "always" ($\Box \varphi = \neg \Diamond \neg\varphi$) are derived.

Semantics: The truth of an LTL formula is evaluated over executions of a finite state machine representing the robot and its environment. An execution is viewed as an infi-

nite sequence of truth assignments to $\pi \in AP$; a formula is satisfiable if it holds for all executions. Informally, the formula $\bigcirc \varphi$ expresses that φ is true in the next "step" or position in the sequence, and the formula $\varphi_1 \mathcal{U} \varphi_2$ expresses the property that φ_1 is true until φ_2 becomes true, and is true only if φ_2 does eventually become true (strong until). The (infinite) truth assignment sequence σ satisfies $\Box \varphi$ if φ is true in every position of the sequence, and satisfies $\Diamond \varphi$ if φ is true at some position of the sequence. Sequence σ satisfies the formula $\Box \Diamond \varphi$ if φ is true infinitely often. For a formal definition of the semantics of LTL, the reader is referred to [19].

LTL is appropriate for specifying robotic behaviors because it provides the ability to describe changes in the truth values of propositions over time. However, other specification languages may be desirable for non-expert users, who will often be unfamiliar with LTL [25, 16, 15, 46]. LTLMoP includes a parser that automatically translates English sentences belonging to a defined grammar [25] into LTL formulas; the grammar includes reactive conditionals, repeated goals, and non-projective locative prepositions such as "between" and "within". This allows users to define desired robot behaviors (including reactive behaviors, e.g., "if you find the target, switch to a hiding role") and specify assumptions about the behavior of the environment (e.g., "the target will never be found in the kitchen") using an intuitive descriptive language rather than the underlying formalism. There are two primary types of properties allowed in a specification – *safety* properties, which guarantee that "something bad never happens", and *liveness* conditions, which state that "something good (eventually) happens". These correspond naturally to LTL formulas with operators "always" ($\Box$) and "eventually" ($\Diamond$).

3.2 Discrete Abstraction

Figure 3.1 shows the three stages of the workspace abstraction for the "hide-and-seek" scenario, from the real environment to a set of convex polygons, and then as a graph with edges connecting adjacent regions. In the discrete abstraction of the problem, the continuous reactive behavior of a robot is described in terms of a finite set of propositions consisting of:

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

The set of sensor propositions (controlled by the environment) are denoted by $\mathcal{X}$, and the set of action and location (i.e., robot-controlled) propositions by $\mathcal{Y}$. In Example 1, $\mathcal{X} = \{\pi_{whistle},\ \pi_{found\_target},\ \pi_{been\_found}\}$, $\mathcal{Y} = \{\pi_{porch}, \pi_{deck}, \pi_{bedroom}, \pi_{dining}, \pi_{living}, \pi_{kitchen}, \pi_{hiding}, \pi_{seeking}, \pi_{counting}\}$. Propositions π_{found_target} and π_{been_found} are true when the robot senses that it has found the other player and been found respectively, π_{hiding}, $\pi_{seeking}$, and $\pi_{counting}$ are true depending on the robot's current role in the game.

The value of each proposition can be thought of as the binary output of a low-level black box component (e.g., $\pi_{whistle}$ could be set based on a threshold on the output of a sensor, $\pi_{bedroom}$ is set based on a localization component, etc.). Sensor inputs are assumed to be reliable in that the binary values are correct, and any uncertainty is dealt with by the low-level algorithms. Similarly, actions are assumed to be reliably implemented once issued (so "counting" never fails), and do not have continuous timing constraints. Location propositions correspond to convex polygons that partition the

workspace[1] (e.g., the proposition $\pi_{bedroom}$ is true if and only if the robot is in the polygon representing the bedroom). Additionally, the formula $\varphi_l = \pi_l \bigwedge_{l' \neq l} \neg \pi_{l'}$ indicates that the robot is in location l and not in any other location (i.e., locations are mutually exclusive).

The possible motion of the robot in the workspace based on the adjacency of the regions is automatically encoded as part of the specification. Legal transitions between adjacent regions are represented as edges between vertices in a graph (with implicit self-loops), and then encoded into a formula over location propositions to appropriately constrain the possible motions of the robot. Valid transitions are specified using a formula φ_{trans}; in the above example,

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

3.3 Controller Synthesis Overview

Figure 3.2 provides an overview of the controller synthesis procedure. The framework handles a class of specifications corresponding to the Generalized Reactivity (1) (GR(1)) fragment of Linear Temporal Logic [37, 53], which captures a large number of high-level tasks specified in practice. A user-defined specification and description of the environment topology is automatically parsed into a formula of the form $\varphi = (\varphi_e \Rightarrow$

[1]Location propositions can correspond to any discrete abstraction of the workspace; this work uses a convex partition.

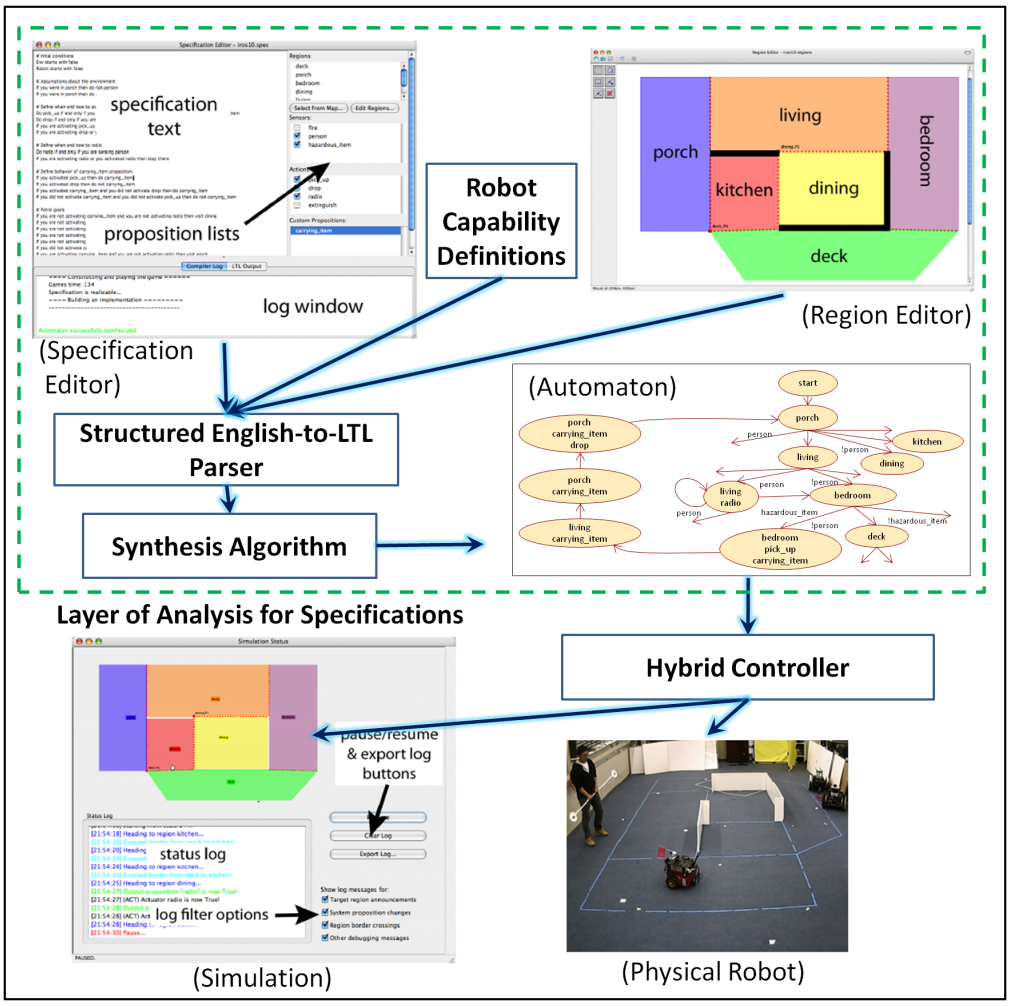


Figure 3.2: Controller synthesis overview [47]

φ_s), where φ_e encodes any assumptions about the sensor propositions, and thus about the behavior of the environment, and φ_s represents the desired behavior of the robot. φ_e and φ_s in turn have the structure $\varphi_e = \varphi_e^i \wedge \varphi_e^t \wedge \varphi_e^g$, $\varphi_s = \varphi_s^i \wedge \varphi_s^t \wedge \varphi_s^g$, where

- φ_e^i and φ_s^i are non-temporal Boolean formulas constraining the initial sensor and robot proposition values respectively.
- φ_e^t represents user-defined assumptions about possible behaviors of the environment, and consists of a conjunction of formulas of the form $\square\, A^i$ where each A^i is a Boolean formula with sub-formulas in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc\, \mathcal{X}$, where $\bigcirc\, \mathcal{X} = \{\bigcirc\, x_1, ..., \bigcirc\, x_n\}$. Intu-

itively, formula φ_e^t constrains the next sensor values $\bigcirc \mathcal{X}$ based on the current sensor $\mathcal{X}$ and robot $\mathcal{Y}$ values. Similarly, φ_s^t represents the robot's required behavior (safety constraints); it consists of a conjunction of formulas of the form $\square A^i$ where each A^i is a Boolean formula in $\mathcal{X} \cup \mathcal{Y} \cup \bigcirc \mathcal{X} \cup \bigcirc \mathcal{Y}$ (the robot's next state can depend on the environment's current and next states). φ_s^t contains φ_{trans} as a subformula.

- φ_e^g and φ_s^g represent assumptions on the environment, and desired goal behaviors for the robot respectively. Both formulas consist of a conjunction of formulas of the form $\square \diamondsuit B^i$ where each B^i is a Boolean formula in $\mathcal{X} \cup \mathcal{Y}$.

In viewing these formulas as corresponding to robot and environment properties, this thesis refers to φ_s^t and φ_e^t as *safety* properties, and φ_s^g and φ_e^g as *liveness* properties. Listing 2 provides the LTL translation of each sentence of the specification in Listing 1, and identifies the corresponding component of the resulting formula φ.

Since the robot can be in exactly one location at any given time, the formula $\varphi_r = \pi_r \wedge \bigwedge_{r' \neq r} \neg \pi_{r'}$ is used to represent the robot being in region r. The robot's motion in the workspace is governed by adjacency of regions, and the availability of controllers to drive it between adjacent regions. In LTLMoP, the adjacency relation is automatically encoded as a logic formula φ_{trans} (see Section 6.1 for an example).

An LTL formula φ is *realizable* if there exists a finite state strategy that, for every finite sequence of truth assignments to the sensor propositions, provides an assignment to the robot propositions such that every infinite sequence of truth assignments to both sets of propositions generated in this manner satisfies φ. The synthesis problem is to find a finite state automaton that encodes this strategy, i.e. whose executions correspond to sequences of truth assignments that satisfy φ.

Definition 1. A *finite state automaton* is a tuple $A = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_{\mathcal{X}}, \gamma_{\mathcal{Y}})$ where

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

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

- Q is a finite set of states.
- $Q_0 \subseteq Q$ is a set of initial states.
- $\mathcal{X}$ is a set of inputs (sensor propositions).
- $\mathcal{Y}$ is a set of outputs (location and action propositions).
- $\delta : Q \times 2^{\mathcal{X}} \rightarrow 2^Q$ is the transition relation. In this work, automata are restricted to be non-blocking, i.e. $\delta(q, x) \neq \emptyset$ for every $q \in Q, x \in 2^{\mathcal{X}}$.
- $\gamma_{\mathcal{X}} : Q \rightarrow 2^{\mathcal{X}}$ is a transition labeling, which associates with each state the set of environment propositions that are true over incoming transitions for that state (note that this set is the same for all transitions into a given state). Note that if $q' \in \delta(q, x)$ then $\gamma_{\mathcal{X}}(q') = x$.
- $\gamma_{\mathcal{Y}} : Q \rightarrow 2^{\mathcal{Y}}$ is a state labeling, associating with each state the set of system propositions true in that state.

Define $\gamma(q) = \gamma_{\mathcal{X}}(q) \cup \gamma_{\mathcal{Y}}(q)$ for $q \in Q$; intuitively, this labels each state with the input and output propositions that are true when the robot is in that state. Given a sequence of states $\sigma = q_0 q_1 q_2 ...$ where $q_0 \in Q_0$, define a sequence-labeling $\Gamma(\sigma) = \gamma(q_0)\gamma(q_1)\gamma(q_2)....$ An automaton is *deterministic* if, for every $q \in Q$ and every $x \in 2^{\mathcal{X}}$, $|\delta(q, x)| = 1$. Unless mentioned explicitly, all automata considered in this work are deterministic. A deterministic automaton corresponds to a robot strategy, as described below.

Let $\delta(q) = \{\delta(q, x) \mid x \in 2^{\mathcal{X}}\}$ denote the set of possible successor states of state q. Finally, let $\delta_{\mathcal{Y}}(q, x) = \gamma_{\mathcal{Y}}(\delta(q, x))$; intuitively, this denotes the actions the robot takes when it is in state q and it senses input x.

Definition 2. Given $\varphi = (\varphi_e \Rightarrow \varphi_s)$, deterministic automaton $A_\varphi = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_{\mathcal{X}}, \gamma_{\mathcal{Y}})$ *realizes* φ if $\forall \sigma = q_0 q_1 q_2 ... \in Q^\omega$ such that $q_0 \in Q_0$ and

$q_{i+1} \in \delta(q_i), \Gamma(\sigma) \models \varphi$.

Given a specification φ, consider the most general nondeterministic automaton over $\mathcal{X}$ and $\mathcal{Y}$, namely $N_\varphi = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_\mathcal{X}, \gamma_\mathcal{Y})$ such that:

- $Q = 2^{\mathcal{X} \cup \mathcal{Y}}$
- for every $q \in Q$, $\delta(q, x) = \{q' \mid \Gamma(qq')True^\omega \models \varphi_e^t \Rightarrow \varphi_s^t\}$

Here $\Gamma(qq')True^\omega$ is the most permissive infinite completion of the finite truth assignment sequence $\Gamma(qq')$, i.e. the infinite sequence that satisfies all LTL formulas from step 2 onward.

Finding a strategy for the robot that fulfills the specification φ can be thought of as exploring the above nondeterministic automaton N_φ in the aim of finding a deterministic automaton A_φ "contained" in it, that realizes the specification. Synthesizing such a deterministic automaton that realizes an arbitrary LTL formula is doubly exponential in the size of the formula [5]. When restricted to LTL formulas of the form $\varphi_e \Rightarrow \varphi_s$ described above, the algorithm introduced in [37] permits synthesis in time polynomial in the size of the state space, assuming specifications are well-separated [44]. Specifications in this fragment of LTL are expressive enough to cover a wide range of typical robot high-level behaviors.

When a specification is realizable, synthesis yields an automaton that implements the specification in a discrete abstraction of the problem. If no such automaton exists, the user is presented with information about the cause of the unrealizability [47, 48, 38]. Successful synthesis enables the construction of a hybrid controller H_{A_φ} that produces the desirable high-level, autonomous robot behavior in the continuous domain. Figure 3.3 shows an example of a synthesized automaton for the hide-and-seek problem, using

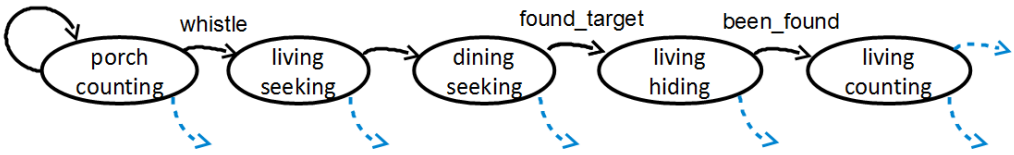


Figure 3.3: Excerpt of "hide-and-seek" automaton

a modified version of the specification in Listing 1, discussed in Section 4.5. Each state of the automaton is labeled by the location and action propositions that are true in that state, and each transition is labeled with sensor propositions that must be true for that transition to be enabled. A transition between two states is achieved by the activation of one or more low-level continuous controllers corresponding to each robot proposition. The atomic controllers used satisfy the bisimulation property [3], which ensures that every change in the discrete robot model can be implemented in the continuous domain (e.g., the motion controllers are guaranteed to drive the robot from one region to another regardless of the initial state within the region). The feedback controllers presented in [10] and [17] are among several that satisfy this property.

The reader is referred to [37] and [26] for more details of the synthesis procedure, and to [26, 11] for a description of how the extracted discrete automaton is transformed into low-level robot control.

3.4 Environment Counterstrategy

When controller synthesis fails the specification is called *unsynthesizable*. Unsynthesizable specifications are either *unsatisfiable*, in which case the robot cannot succeed no matter what happens in the environment (e.g., if the task requires patrolling a disconnected workspace), or *unrealizable*, in which case there exists at least one environment that can prevent the desired behavior (e.g., if in the above task, the environment can dis-

connect an otherwise connected workspace, such as by closing a door). More examples illustrating the two cases can be found in Chapter 4.

In either case, the robot can fail in one of two ways: either it ends up in a state from which it has no valid moves (termed *deadlock*), or the robot is able to change its state infinitely, but one of its goals is unreachable without violating the specified safety requirements (termed *livelock*). In the context of unsatisfiability, an example of deadlock is when the system safety conditions contain a contradiction within themselves. Similarly, unrealizable deadlock occurs when the environment has at least one strategy for forcing the system into a deadlocked state. Livelock occurs when there is one or more goals that cannot be reached while still following the given safety conditions.

In the case of unsynthesizable specifications, the counterstrategy synthesis algorithm introduced in [39] can be used to extract a strategy for the environment, which provides sequences of environment inputs that prevent the specified robot behavior.

Definition 3. An *environment counterstrategy* for LTL formula φ is a tuple $A_{\varphi}^{e} = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta_e, \delta_s, \gamma_{\mathcal{X}}, \gamma_{\mathcal{Y}}, \gamma_{goals})$, where

- Q is a set of states.
- $Q_0 \subseteq Q$ is a set of initial states.
- $\mathcal{X}$ is a set of inputs (sensor propositions).
- $\mathcal{Y}$ is a set of outputs (location and action propositions).
- $\delta_e : Q \rightarrow 2^{\mathcal{X}}$ is the deterministic input transition relation, which provides the input propositions that are true in the next time step given the current state q, and satisfies φ_e^t.
- $\delta_s : Q \times 2^{\mathcal{X}} \rightarrow 2^{Q}$ is the (nondeterministic) robot transition relation. If $\delta_s(q, x) = \emptyset$ for some $x \in 2^{\mathcal{X}}, q \in Q$, then there is no next-step assignment

to the set of outputs that satisfies the robot's transition relation φ_s^t, given the next set of environment inputs x and the current state q.

- $\gamma_{\mathcal{X}} : Q \rightarrow 2^{\mathcal{X}}$ is a transition labeling, which associates with each state the set of environment propositions that are true over incoming transitions for that state (note that this set is the same for all transitions into a given state). Note that if $q' \in \delta_s(q, x)$ then $\gamma_{\mathcal{X}}(q') = x$.
- $\gamma_{\mathcal{Y}} : Q \rightarrow 2^{\mathcal{Y}}$ is a state labeling, associating with each state the set of robot propositions true in that state.
- $\gamma_{goals} : Q \rightarrow \mathbb{Z}^+$ labels each state with the index of a robot goal that is prevented by that state. During the counterstrategy extraction, every state in the counterstrategy is marked with some robot goal[39].

The counterstrategy provides truth assignments to the input propositions (according to the transition function δ_e) that cause either livelock or deadlock. The inputs provided by δ_e in each state satisfy φ_e^t, meaning that for all $q \in Q$, the truth assignment sequence $(\gamma_{\mathcal{X}}(q) \cup \gamma_{\mathcal{Y}}(q), \gamma_{\mathcal{X}}(\delta_e(q)))$ satisfies A_i for each conjunct A_i in φ_e^t (note that A_i is a formula over two consecutive time steps).

CHAPTER 4

EXPLAINING IMPOSSIBLE HIGH-LEVEL ROBOT BEHAVIORS

This chapter describes an algorithm for automatically analyzing an LTL specification belonging to the class GR(1), to identify and focus the user's attention on relevant portions thereof. The goal is to enable iterated specification analysis and modification, and facilitate construction of a controller that achieves the user-intended behavior. By the completeness of the synthesis algorithm in [37], when a specification is unsynthesizable, there exists an admissible adversarial environment strategy that demonstrates the system's failure to achieve the specified behavior. This environment strategy is referred to as the counterstrategy, following [39]. Counterstrategy generation is leveraged, along with other tools like Boolean satisfiability testing, to provide explicit feedback on unsynthesizable specifications in the robot control domain. Feedback is provided by emphasizing problematic parts of the user-defined specification (either desired robot behavior or environment assumptions), and identifying cases of unexpected and undesirable behavior such as the trivial solutions mentioned earlier. In addition, the specification designer can infer logical inconsistencies and other reasons for unsynthesizability by interacting with an environment counter-strategy simulation, which demonstrates the precise environment actions that thwart the robot.

The chapter is structured as follows. Section 4.1 provides a formal problem statement. Section 4.2 describes the types of unsynthesizability handled by this work, and provides illustrative examples. Sections 4.3 and 4.4 contain the main contributions: an algorithm for identifying unsatisfiable or unrealizable components of a specification, and an interactive game for exploring reasons for failure in unrealizable specifications; both sections include examples demonstrating the corresponding implementations in LTLMoP. The chapter concludes with a description of open challenges in Section 4.5.

4.1 Problem Statement

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

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

4.2 Unsynthesizable Specifications and Undesirable Behavior

The specification in Listing 2 is unsynthesizable, and in particular it is unrealizable, because an adversarial environment can force the robot into a safety violation by setting π_{found_target} to true and $\pi_{whistle}$ to false when $\pi_{counting}$ is true; the robot will turn on its hiding behavior in the next time step ($\bigcirc \pi_{hiding}$ is set by line 10), but still be counting ($\bigcirc \pi_{counting}$ required by line 11), and therefore simultaneously satisfy $\bigcirc \pi_{hiding} \wedge \bigcirc \pi_{counting}$, violating the safety condition in line 7.

4.2.1 Unsynthesizable Categories

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

Unsatisfiability

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

Always not **porch** $\square \neg \bigcirc \varphi_{porch}$ (in φ_s^t)

Visit **porch** $\square \diamondsuit \varphi_{porch}$ (in φ_s^g)

This specification is not synthesizable, and in particular it is unsatisfiable, since φ_s^t and φ_s^g are inconsistent no matter what the environment does, and so the robot cannot win.

Unrealizability

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

If you are sensing **whistle** then do **porch**

$\square(\pi_{whistle} \Rightarrow \bigcirc \varphi_{porch})$ (in φ_s^t)

Depending on the current location, φ_{trans} does not always allow $\bigcirc \varphi_{porch}$. For example, if the robot hears the whistle in the bedroom, it cannot reach the porch in the next discrete step (without passing another region first), so there are no further transitions satisfying the robot safety. The environment can thus win from some initial states (e.g. bedroom) by setting $\pi_{whistle}$ to true. This specification is *unrealizable*, but not *unsatisfiable* – there are environment strategies for which the robot achieves the desired behavior, such as those that never set $\pi_{whistle}$ to true.

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

and the robot would win whether or not it fulfilled its goals.

Undesirable Behavior After Synthesis

Consider the same map again, with the following specification:

Always not **porch**	$\square \neg \bigcirc \varphi_{porch}$	(in φ_s^t)
Visit **porch**	$\square \diamondsuit \varphi_{porch}$	(in φ_s^g)
Always sense **whistle** and		
do not sense **whistle**	$\square(\bigcirc \pi_{whistle} \wedge \bigcirc \neg\pi_{whistle})$	(in φ_e^t)

Here φ_e^t is unsatisfiable, so φ_e is unsatisfiable. Since the antecedent of the implication is always false, the formula $\varphi_e \implies \varphi_s$ is satisfied by any automaton, and all initial states are winning for the robot, even though the robot itself is also unsatisfiable. The algorithm in [37] returns a trivial automaton consisting of all the initial states, but no transitions between states; in the case of the hide-and-seek example, this is a single state, in which the robot is counting in the porch. Since each state in the automaton has an implicit self-loop in the continuous level implementation, a controller based on this automaton would cause the robot to stay in the porch indefinitely – this is likely not the user-intended behavior.

4.2.2 Causes of Failure

Section 4.2.1 detailed the distinction between unsatisfiable and unrealizable specifications. In either case, failure occurs either if it is possible for the environment to steer the corresponding two-player game into a state from which the robot has no valid move (as in Example 4.2.1), or if one of the robot's liveness conditions (goals) cannot be reached (as in Example 4.2.1). The former case is termed *deadlock*, and the latter case *livelock*.

4.2.3 Identifying Deadlock

Identifying deadlock calls for a characterization of the set of "bad" states from which the environment can force the robot into a state such that every transition will violate the robot safety (in which case the robot has no next move in the game in [37]). Such a characterization of states can be expressed in the modal μ-calculus, which extends propositional modal logic with least and greatest fixpoint operators μ, ν [18]. The μ-calculus over game structures is defined as in [37]. A formula φ is interpreted as the set of states $[\![\varphi]\!]$ in which φ is true. Under this interpretation, the logical operator $\diamondsuit$ is defined such that a state s is included in $[\![\diamondsuit\, \varphi]\!]$ if the robot can force play to reach a state in $[\![\varphi]\!]$, regardless of how the environment moves from s. For example, if the environment safety condition includes "If you were in porch then do not whistle", then any state where the robot is in the porch is included in $[\![\diamondsuit\, \neg\pi_{whistle}]\!]$, since the environment cannot activate $\pi_{whistle}$ in the next state. Similarly, operator $\boxdot$ is defined such that a state s is included in $[\![\boxdot\, \varphi]\!]$ if the environment can force play to reach a state in $[\![\varphi]\!]$.

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

4.3 Algorithm for Analysis of Specifications

This section describes in detail the steps of Algorithm 1 introduced in [48], for isolating sources of unsatisfiability and unrealizability in the robot and environment components of an unsynthesizable or trivial specification. Given an input specification parsed into a suitable representation of the environment (φ_e) and robot (φ_s) LTL formulas, properties of the synthesis problem are leveraged to determine whether each of φ_e and φ_s is unrealizable or unsatisfiable, and present the user with this information. In LTLMoP, the presented algorithm is implemented in the JTLV framework [6], with the corresponding formulas for the initial conditions, transitions and goals represented as Binary Decision Diagrams (BDDs) [14]. The BDD representation of a formula is a directed acyclic graph representing the set of proposition value combinations that satisfy it; BDDs enable efficient operations on formulas.

4.3.1 Synthesis and Trivial Automata

The following pseudocode describes the initialization of variables from Algorithm 1 in [48]. If the specification φ is realizable, a BDD representation of the set of all implementing control automata AUT_SET is synthesized using the $SYNTHESIS$ algorithm from [37], and a single such automaton AUT is extracted. Note that in the BDD representation, **FALSE** denotes the empty set, and **TRUE** denotes the set of *all* automata. Otherwise, a set of all possible counterstrategies CTR_SET is obtained following a construction $COUNTERSTRATEGY$, adapted from that presented in [39]; the counterstrategy generation algorithm for GR(1), like synthesis, also runs in time polynomial in the size of the state space. If an automaton is synthesized, but has no transitions (i.e. is trivial), the user is alerted to this fact.

Algorithm 1 Initialization of variables from Algorithm 1 in [48]

$\varphi_p^t = \bigwedge_j \square A_p^j, \varphi_p^g = \bigwedge_{i=1}^{n_p^g} \square \diamondsuit B_p^i$ for $p \in \{s, e\}$
$AUT\_SET \leftarrow SYNTHESIS(s, e)$
if AUT_SET ! = **FALSE** (spec. is synthesizable) **then**
    $AUT \leftarrow AUT\_SET$
    if AUT has no transitions **then**
        flag as *trivial*
else
    $CTR\_SET \leftarrow COUNTERSTRATEGY(s, e)$

4.3.2 Unsatisfiable Initial Conditions and Transition Relations

Recall that φ_e^t and φ_s^t consist of a conjunction of formulas of the form $\square A^i$ where each A^i is a Boolean formula, so for either of these, an emptiness check on the BDD representing the set of variable assignments satisfying $\varphi_p^i \wedge \bigwedge_i A^i$ determines whether the transitions in a single time step are satisfiable from the initial condition. The pseudocode in Algorithm 2 checks for the unsatisfiability of the initial conditions and the transitions relation (safety) for both environment and robot.

Algorithm 2 Initial conditions and transition unsatisfiability tests from Algorithm 1 in [48]

if φ_p^i == **FALSE then**
    player p has *unsatisfiable initial conditions*
if $\varphi_p^i \wedge \bigwedge_i A_p^i$ == **FALSE then**
    p has *unsatisfiable transitions*

The above check will not identify unsatisfiability of following the transitions over multiple time steps; for example, the transition relation $\square(\pi_{hiding} \implies \bigcirc \pi_{hiding}) \wedge \square(\pi_{hiding} \implies \neg \bigcirc \pi_{hiding}) \wedge \square(\neg\pi_{hiding} \implies \bigcirc \pi_{hiding})$ is unsatisfiable when starting from the initial condition $\neg\pi_{hiding}$, because π_{hiding} is true in the second time step leading to no valid transitions (since any valid transition would have to satisfy both π_{hiding} and $\neg\pi_{hiding}$); however the analysis so far will not detect this. Such "multi-step" unsatisfiability of the transitions is identified by computing the set of environment coun-

terstrategies (i.e. the strategies the environment can use to find sensor inputs such that there is no robot response fulfilling the specification), using the counterstrategy synthesis algorithm in [39]. If every sequence of environment moves is in this counterstrategy, then the robot must be unsatisfiable. In addition, if every sequence of environment moves forces the robot into deadlock (rather than livelock), the robot safety is unsatisfiable; this is identified using a fixpoint computation as described earlier. The symmetric case for multi-step unsatisfiable environment transitions looks at the set of robot winning strategies and checks that every sequence of robot actions is winning.

Algorithm 3 Multi-step unsatisfiability tests from Algorithm 1 in [48]

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

4.3.3 Unsatisfiable Goals

The next steps of the algorithm check for unsatisfiability of robot and environment liveness conditions. Any liveness condition φ_p^g consists of a conjunction of clauses of the

Algorithm 4 Unsatisfiable goal tests from Algorithm 1 in [48]

15: **for** $i := 1$ to n_p^g **do**
16: **if** $B_p^i == \textbf{FALSE}$ **then**
17: p *goal* i *is unsatisfiable*
18: **for** $i := 1$ to n_p^g **do**
19: **if** $B_p^i \wedge \bigwedge_j A_p^j == \textbf{FALSE}$ **OR** $\bigcirc B_p^i \wedge \bigwedge_j A_p^j == \textbf{FALSE}$ **then**
20: p *is unsatisfiable between goal* i *and transitions*

form $\square \diamondsuit B^i$, and the safety φ_p^t consists of a conjunction of formulas of the form $\square A^j$. So a contradiction in $\varphi_p^g \wedge \varphi_p^t$ implies that some $\square \diamondsuit B^i$ is inconsistent with $\bigwedge_j \square A^j$, i.e., $\diamondsuit B^i$ is inconsistent with $\bigwedge_j A^j$. The A^js in the transition formula govern proposition values in the current and next time steps (as described in Section 3.3), and in order for a goal B^i to be satisfied infinitely often, it needs to be consistent with each A^j in

both the current and next time steps (so there are valid transitions into and out of each goal state). This is confirmed by checking $B^i \wedge \bigwedge_j A^j$ and $\bigcirc B^i \wedge \bigwedge_j A^j$ for consistency – if either is inconsistent, then liveness condition B^i cannot be fulfilled infinitely often while following the transitions allowed by the safety.

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

Algorithm 5 Multi-step unsatisfiable and unrealizable goal tests from Algorithm 1 in [48]

23: **for** $i := 1$ to n_s^g **do**
24: $\quad \varphi_{s_i}^g = \bigwedge_{k=1}^{i} \square \diamondsuit B^{s_k}, \varphi_{s_i}^t = \varphi_s^t, \varphi_{s_i}^i = \varphi_s^i$
25: $\quad AUT\_SET_i \leftarrow SYNTHESIS(s_i, e)$
26: $\quad$ **if** AUT_SET_i == **FALSE** (unsynthesizable) **then**
27: $\quad\quad CTR\_SET_i \leftarrow COUNTERSTRATEGY(s_i, e)$
28: $\quad\quad$ **if** CTR_SET_i == **TRUE then**
29: $\quad\quad\quad i^{th}$ *robot goal inconsistent with transition relation*
30: $\quad\quad$ **else if** AUT_SET_{i-1} ! = **FALSE then**
31: $\quad\quad\quad i^{th}$ *robot goal is unrealizable*
32: **for** $i := n_e^g$ to 1 **do**
33: $\quad \varphi_{e_i}^g = \bigwedge_{k=i}^{n_e^g} \square \diamondsuit B^{e_k}, \varphi_{e_i}^t = \varphi_e^t, \varphi_{e_i}^i = \varphi_e^i$
34: $\quad AUT\_SET_i \leftarrow SYNTHESIS(s, e_i)$
35: $\quad$ **if** AUT_SET_i ! = **FALSE** (synthesizable) **then**
36: $\quad\quad$ **if** AUT_SET_i == **TRUE then**
37: $\quad\quad\quad i^{th}$ *environment liveness inconsistent with transitions*
38: $\quad$ **else if** AUT_SET_{i+1} == **FALSE then**
39: $\quad\quad i^{th}$ *environment liveness condition is unrealizable*

If the environment counterstrategy is **TRUE**, then the robot is in fact unsatisfiable due to the most recently added goal (lines 28-29). A symmetric test for the environ-

ment runs the synthesis algorithm starting with all environment liveness conditions, and removes them one by one (lines 32-39). Similarly, if every robot strategy is winning, the current environment goals must be unsatisfiable, and if removing an environment liveness condition makes the specification unsynthesizable, the robot's winning strategy involved falsifying the removed environment liveness (36-37).

If the algorithm does not detect robot unsatisfiability, the robot might still be unrealizable. To win from an initial state of the game, the environment must either force the robot into deadlock or livelock. As described earlier, reachability analysis using fixpoint operators suffices to check for a sequence of environment actions forcing the robot into deadlock, and likewise for the robot forcing the environment into deadlock, as in Algorithm 6.

Algorithm 6 Deadlock tests from Algorithm 1 in [48]

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

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

Example 2. Consider again the specification in Listing 1. As mentioned in Chapter 3, this specification fails to produce a controller, and it is difficult to pinpoint the problem without the presented analysis. The proposed tests determine that the robot is unrealizable because the environment can force a safety violation, allowing the user to focus their attention on the relevant sentences (highlighted as in Figure 4.1).

4.3.4 Guarantees

The algorithm presented above is sound and complete for robot unsynthesizability, in the sense that every incidence of robot unsatisfiability or unrealizability falls into one of the handled cases. Note that the algorithm provides information about both robot and environment components. By notifying the user if the environment is unsatisfiable or unrealizable, they are alerted to the fact that the behavior generated may not be as intended, prior to execution. However, there are cases of environment unsatisfiability or unrealizability that may not be identified by the above tests. When the environment is unrealizable or unsatisfiable because of livelock, but the robot itself is deadlocked, the robot has no infinite strategies, and therefore cannot cause environment livelock. On the other hand, if the robot is realizable independent of the environment, the tests in Algorithm 5 will not reveal any information about the environment, since synthesis will never fail. However in this case, following the robot strategy construction in [37], the robot will achieve the desired behavior rather than prevent the environment from fulfilling its goals, so the environment unrealizability has no consequences for the robot's behavior. All other cases of environment unrealizability and unsatisfiability are captured by the above tests.

4.3.5 Complexity

Algorithms 1 and 5 are polynomial in the size of the state space. This follows because computing the set of implementing automata or environment counterstrategies is polynomial in the state space [37, 39], and the number of times these subcomponents are repeated is linear in the number of robot and environment goals respectively. The tests for deadlock in Algorithms 3 and 6 are also polynomial in the state space for the same

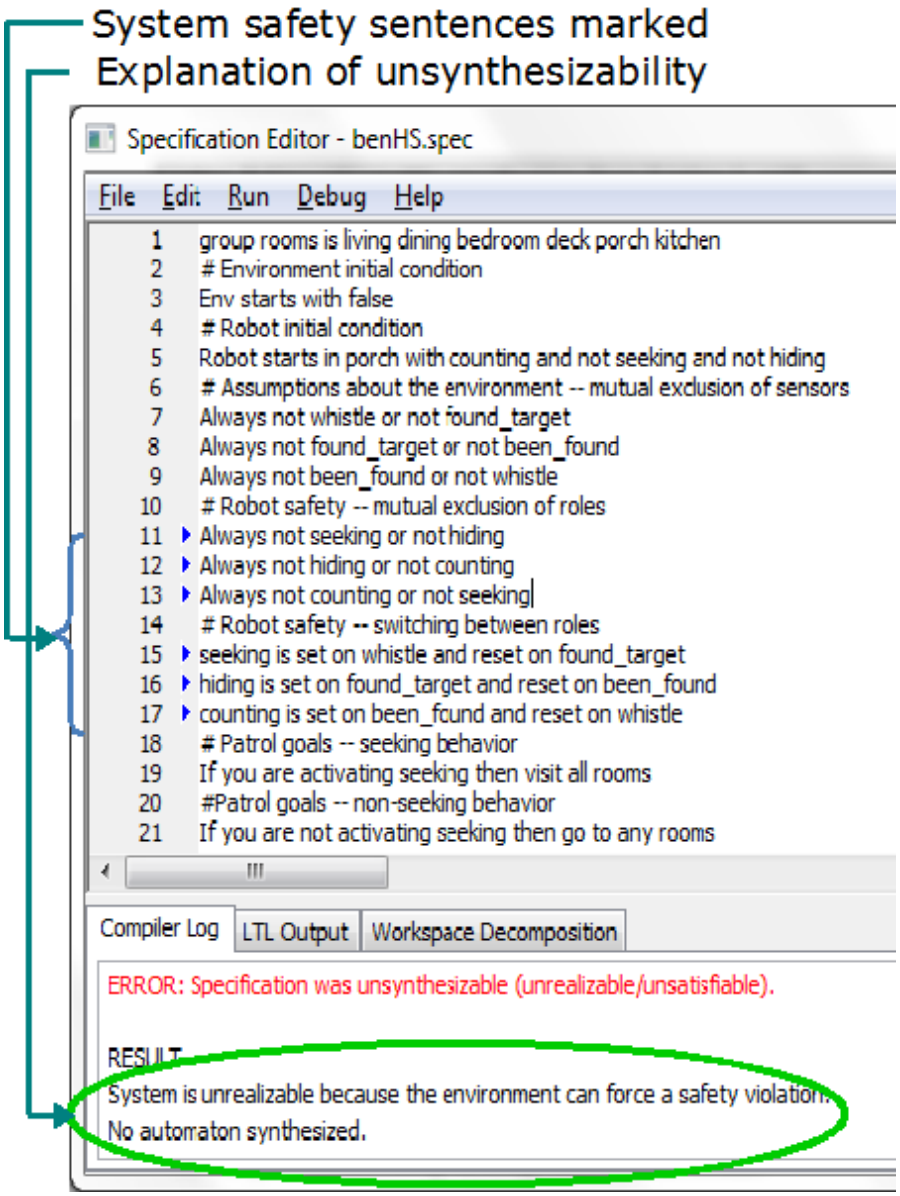


Figure 4.1: Analyzing an unsynthesizable specification

reason. Algorithms 2 and 4 involve conjunctions on BDDs, which are also implemented in polynomial time. The runtime of the proposed algorithm is therefore polynomial in the size of the state space.

4.4 Interactive Exploration of Unrealizable Specifications

The algorithm described in the previous section enables highlighting of sentences of the specification that contribute to the unsynthesizability. However, so far the user is not presented with a demonstration of why the specification cannot be implemented. In the case of deadlock, it may be possible to present the user with a set of finite sequences of moves leading to a safety violation. However, a possibly exponential number of sequences of moves is needed to demonstrate livelock. This problem is addressed in this work via an interactive game. The counterstrategy synthesis algorithm introduced in [39] is used to extract a strategy for the environment, which provides sequences of environment actions such that there is no robot response fulfilling the specification. The user interacts with this strategy by selecting the robot actions and movement in every time step in response to the sensor inputs provided by LTLMoP. The user can change the state of robot actuators by clicking toggle buttons, and select a region to move to by clicking on a map. The available choices are automatically constrained according to the robot safety conditions: forbidden regions are blacked out on the map, and illegal action choices raise an error, as shown in Figure 4.2(d).

Example 3. For the unsynthesizable example from Listing 1, the cause of unrealizability is evident at the very first state, as shown in Figure 4.3. The environment has set π_{found_target} to true, and there are no safe robot actions from the displayed state, as indicated by the error message on the screen.

Example 4. Consider the specification in Listing 3, drawn from the "fire-fighting" scenario introduced in [48]. The robot task is to enter the house depicted in Figure 3.1 from the deck and visit the porch infinitely often. If it encounters a person, it cannot move directly to the kitchen. Similarly, if it senses fire, it cannot move to the living

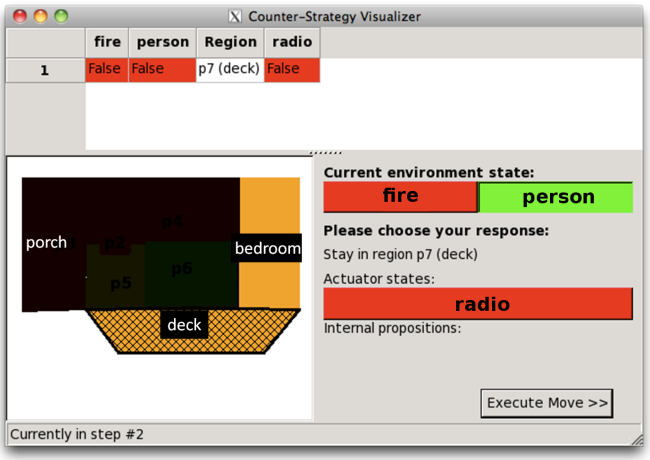
room. The radio is always turned off, and the assumption on the environment is that a person will never be sensed simultaneously with fire.

Listing 3 Example of unrealizable specification for counterstrategy visualization.

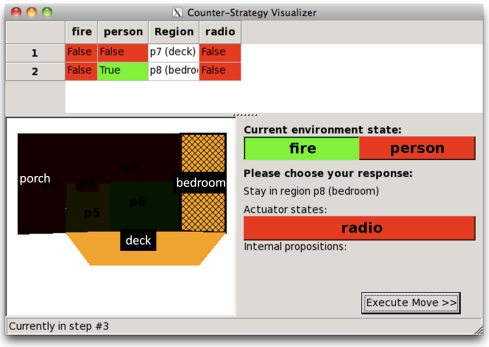
```
Robot starts with false
Robot starts in deck
Visit porch
If you are sensing person then do not kitchen
If you are sensing fire then do not living
Always do not (fire and person)
Always do not radio
```

The environment can prevent the robot from satisfying its goal (to visit the porch infinitely often) by alternately enabling π_{fire} and π_{person}, thereby trapping the robot in the deck and bedroom, i.e. away from the porch. Figure 4.2 shows the first three steps of this (infinite) counterstrategy being played through in the Counterstrategy Visualization Tool. Regions that cannot be chosen due to the motion constraints are blacked out.

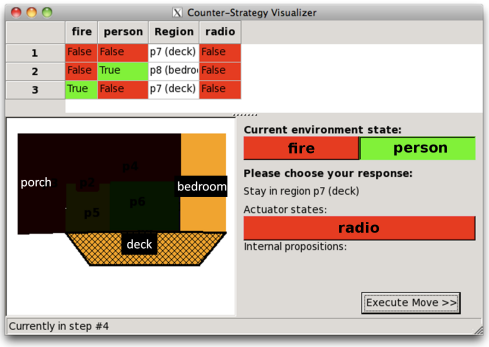
The first step is the setting of a valid initial condition from which the environment can win: the robot is in the deck and all other sensor and action propositions are set to false. Second, the environment enables π_{person} so the robot cannot enter the kitchen and is in the next step confined (by the adjacency graph in Figure 3.1(c)) to the deck and bedroom as depicted in Figure 4.2(a). The user responds by moving the robot to the bedroom, and the environment then switches to enabling π_{fire}, thus disabling π_{person} as assumed in line 6 (Figure 4.2(b)); this prevents the robot from entering the living room, and the user returns to the deck. This move results in the original configuration, and the environment again switches on π_{person} and turns off π_{fire}, as shown in Figure 4.2(c); at this point it should be clear to the user that the environment can keep the robot out of the porch.



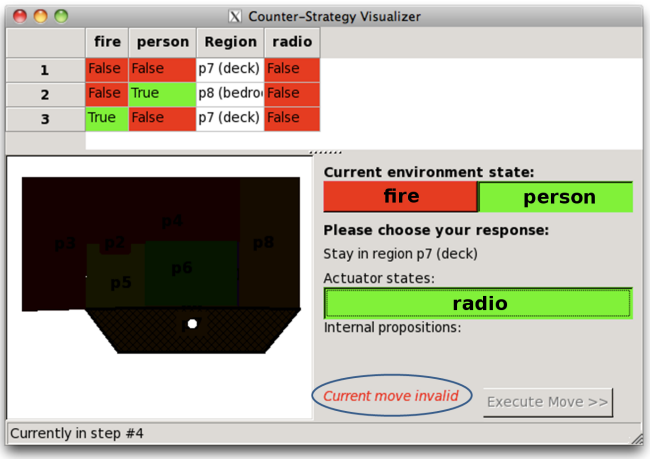
(a) Robot starts in the $deck$, environment's first move turns on $person$.



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



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



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

Figure 4.2: Interactive Game for Unsynthesizable Specifications

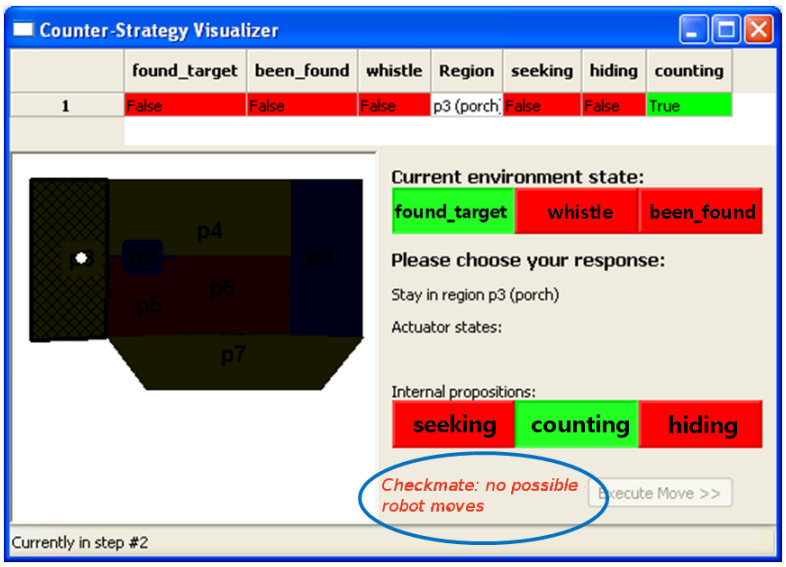


Figure 4.3: Counterstrategy visualization for "hide-and-seek" example. The circled message reads, "Checkmate: no possible robot moves".

4.5 Conclusions

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

Future work will leverage existing techniques to further isolate the source of failure and provide the user with more comprehensive feedback. This includes identifying

of *unsynthesizable cores*, as described in Chapter 5, but also could include suggested modifications to the specification in the form of assumptions to be added. For instance, the specification in Example 3 can be made realizable by adding assumptions on the environment to prevent it from setting π_{found_target} to true and $\pi_{whistle}$ to false when $\pi_{counting}$ is true (additional assumptions are also needed to prevent the environment from causing safety violations when π_{hiding} and $\pi_{seeking}$ are true). A key direction of future research is thus the development of efficient techniques for computing unsatisfiable and unrealizable cores, and added assumptions for specifications in the robotics domain.

CHAPTER 5

UNSYNTHESIZABLE CORES – MINIMAL EXPLANATIONS FOR IMPOSSIBLE HIGH-LEVEL ROBOT BEHAVIORS

The work presented in Chapter 4 identified sub-portions of the specification that contribute to the problem, but left open the challenge of refining this feedback to the finest possible granularity, providing the user with a *minimal* cause of unsynthesizability. The work presented in this chapter builds upon previous analysis to identify *unsynthesizable cores* – minimal subsets of the desired robot behavior that cause it to be unsatisfiable or unrealizable. In addition to the original synthesis algorithm, the analysis makes use of off-the-shelf SAT-solvers and the environment counterstrategy (the adversarial environment strategy that prevents the robot from succeeding) to find this minimal cause of unsynthesizability. This work subsumes that presented in [46, 49], and includes additional techniques for identifying unrealizable cores in certain cases, as described in Section 5.4.

This chapter is structured as follows. Section 5.1 describes types of unsynthesizability, and presents a formal definition of the problem of identifying unsynthesizable cores. Sections 5.2 and 5.3 present techniques for using Boolean satisfiability to identify unsatisfiable and unrealizable cores respectively. Section 5.4 describes an alternative method for identifying cores, based on iterated realizability checks. Section 5.5 demonstrates the effectiveness of the more fine-grained feedback on example specifications. The chapter concludes with a description of future work in Section 5.6.

5.1 Problem Statement

A specification that does not yield an implementing automaton is called *unsynthesizable*. Unsynthesizable specifications are either *unsatisfiable*, in which case the robot cannot succeed no matter what happens in the environment (e.g., if the task requires patrolling a disconnected workspace), or *unrealizable*, in which case there exists at least

one environment that can prevent the desired behavior (e.g., if in the above task, the environment can disconnect an otherwise connected workspace, such as by closing a door). More examples illustrating the two cases can be found in [38].

In either case, the robot can fail in one of two ways: either it ends up in a state from which it has no moves that satisfy the specified safety requirements φ_s^t (this is termed *deadlock*), or the robot is able to change its state infinitely, but one of the goals in φ_s^g is unreachable without violating φ_s^t (termed *livelock*). In the context of unsatisfiability, an example of deadlock is when the system safety conditions contain a contradiction within themselves. Similarly, unrealizable deadlock occurs when the environment has at least one strategy for forcing the system into a deadlocked state. Livelock occurs when there is one or more goals that cannot be reached while still following the given safety conditions.

Consider the specification in Listing 4, in which the robot is operating in the workspace depicted in Figure 5.1. The robot starts at the left hand side of the hallway (1), and must visit the goal on the right (4). The safeties specify that the robot should not pass through region $r5$ if it senses a person (2). Additionally, the robot should always activate its camera (3).

Listing 4 Unrealizable specification – livelock

1. Robot starts in start with camera
   ($\pi_{start} \wedge \pi_{camera}$, part of φ_s^i))
2. If you are sensing a person then do not r5
   ($\square(\bigcirc \pi_{person} \Rightarrow \neg \bigcirc \pi_{r5}$), part of φ_s^t)
3. Always activate the camera ($\square \bigcirc \pi_{camera}$, part of φ_s^t)
4. Visit the goal ($\square \diamondsuit \pi_{goal}$, part of φ_s^g)

It is clear that the environment can prevent the goal in (4) by always activating the "person" sensor (π_{person}), because of the initial condition in (1) and the safety require-

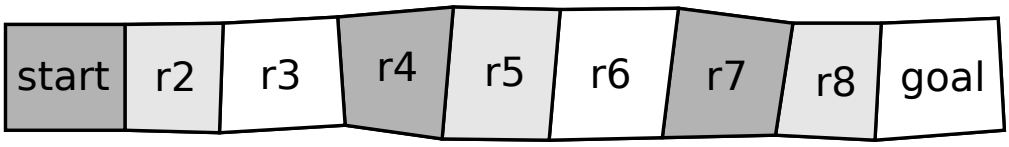


Figure 5.1: Map of robot workspace in Specification 4

ment in (2). Note that Listing 4 is a case of livelock: the robot can follow its safety indefinitely by moving between the first four rooms on the left, but is prevented from ever reaching the goal if it sees a person all the time – the environment is able to disconnect the topology using the person sensor.

Previous work produced explanations of unsynthesizability in terms of combinations of the specification components (i.e., initial conditions, safeties and goals) [38]. However, in many cases, the true conflict lies in small subformulas of these components. For example, the safety requirement (3) in Listing 4 is irrelevant to its unsynthesizability, and should be excluded from any explanation of failure. The specification analysis algorithm presented in [38] will narrow down the cause of unsynthesizability to the goal in (4), but also highlight the entirely of φ_s^t, declaring that the environment can prevent the goal because of some subset of the safeties (without identifying the exact subset).

This motivates the identification of small, minimal, "core" explanations of the unsynthesizability. A first step is to define what is meant by an *unsynthesizable core*. This work draws inspiration from the Boolean satisfiability (SAT) literature to define an unsynthesizable core of a GR(1) LTL formula. Given an unsatisfiable SAT formula in conjunctive normal form (CNF), an unsatisfiable core is traditionally defined as a subset of CNF clauses that is still unsatisfiable. A *minimal* unsatisfiable core is one such that every proper subset is satisfiable; a given SAT formula can have multiple minimal unsatisfiable cores of varying sizes. This definition should be distinguished from that of a *minimum* unsatisfiable core, which is one containing the smallest number of original

clauses that are unsatisfiable in themselves. While there are several practical techniques for computing minimal unsatisfiable cores, and many modern SAT-solvers include this functionality, there are no known practical algorithms for computing minimum cores.

Let $\varphi_1 \preceq \varphi_2$ ($\varphi_1 \prec \varphi_2$) denote that φ_1 is a subformula (strict subformula) of φ_2.

Definition 4. Given a specification $\varphi = \varphi_e \Rightarrow \varphi_s$, a *minimal unsynthesizable core* is a subformula $\varphi_s^* \preceq \varphi_s$ such that $\varphi_e \Rightarrow \varphi_s^*$ is unsynthesizable, and for all $\varphi_s' \prec \varphi_s^*$, $\varphi_e \Rightarrow \varphi_s'$ is synthesizable.

Problem 1. *Given an unsynthesizable formula φ, return a minimal unsynthesizable core $\varphi_s^* \preceq \varphi_s$.*

5.2 Unsatisfiable Cores via Propositional SAT

This section describes how *unsatisfiable* components of the robot specification φ_s are further analyzed to narrow the cause of unsatisfiability, for both deadlock and livelock, using Boolean satisfiability testing. Extending these techniques to the environment assumptions φ_e is straightforward.

The Boolean satisfiability problem, or SAT, is the problem of determining whether there exists a truth assignment to a set of propositions that satisfies a given Boolean formula. A Boolean formula in Conjunctive Normal Form (CNF) is one that has been rewritten as a conjunction of clauses, each of which is a disjunctions of literals, where a literal is a Boolean proposition or its negation. For a Boolean formula in CNF, an unsatisfiable core is defined as a subset of CNF clauses whose conjunction is still unsatisfiable; a minimal unsatisfiable core is one such that removing any clause results in a satisfiable formula.

5.2.1 Unsatisfiable Cores for Deadlock

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

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

where $\varphi[a/b]$ represents φ with all occurrences of subformula a replaced with b. This formula is called the depth-d *unrolling* of φ. In Listing 5, $\psi^1(\varphi_s^t) = \neg\pi^1_{kitchen} \wedge \pi^2_{camera}$ and $\psi^2(\varphi_s^t) = \neg\pi^1_{kitchen} \wedge \pi^2_{camera} \wedge \neg\pi^2_{kitchen} \wedge \pi^3_{camera}$, where $\pi^i_{kitchen}$ is a propositional variable representing the value of $\pi_{kitchen}$ at time step i. Given the depth-d unrolling $\psi^d(\varphi_s^t)$ of the robot safety formula, define $\psi^d_{fromInit} = \varphi_s^i[\pi/\pi^1] \wedge \psi^d(\varphi_s^t)$.

In the case of deadlock, which can be identified as in [48, 38], a series of Boolean formulas $\{\psi^d_{fromInit}\}$ is produced by incrementally unrolling the robot safety formula φ_s^t, and the satisfiability of $\psi^d_{fromInit}$ is checked at each depth. To perform this check, the formula $\psi^d_{fromInit}$ is first converted into CNF, so that it can be provided as input to an off-the-shelf SAT-solver; this work uses PicoSAT [8]. If $\psi^d_{fromInit}$ is found unsatisfiable, there is no valid sequence of actions that follow the robot safety condition for d time steps starting from the initial condition. In this case, the SAT solver returns a minimal unsatisfiable subformula, in the form of a subset of the CNF clauses.

When translating the Boolean formula $\psi^d_{fromInit}$ to CNF, a mapping is maintained between the portions of the the original specification, and the clauses they generate. This enables the CNF minimal unsatisfiable core to be traced back to the corresponding safety conjuncts and initial conditions in the specification.

Listing 5 is a deadlocked specification, referring to a robot operating in the workspace depicted in Figure 5.2. The described method begins at the initial state

Listing 5 Core-finding example – unsatisfiable deadlock

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

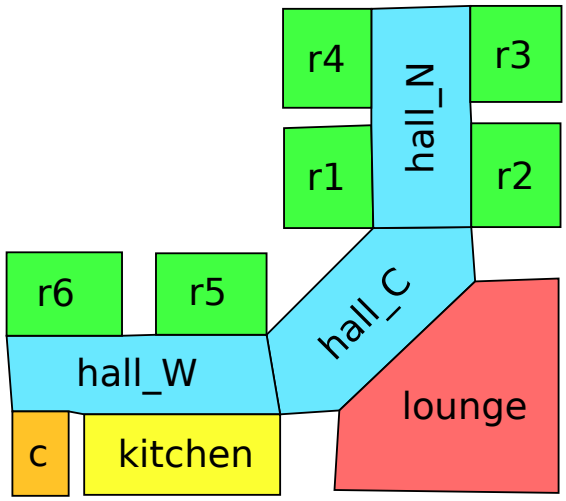


Figure 5.2: Map of hospital workspace ("c" is the closet)

described by φ_s^i (lines 1 and 2), and unrolls it to $\psi^1_{fromInit} = \pi^1_{kitchen} \wedge \neg\pi^1_{kitchen} \wedge \neg\pi^1_{kitchen} \wedge \pi^2_{camera}$ above. Note that $\varphi^1_{fromInit}$ is already unsatisfiable, and the core is given by the subformula $\pi^1_{kitchen} \wedge \neg\pi^1_{kitchen}$, which in turn maps back to lines 1 and 2. This is because the two statements combined require the robot to both start in the kitchen and not start in the kitchen. Section 5.5 contains another example demonstrating unsatisfiable core-finding for deadlock.

5.2.2 Unsatisfiable Cores for Livelock

In the case of livelock, a similar unrolling procedure can be applied to determine the core set of clauses that prevent a goal from being fulfilled. A propositional formula

is generated by unrolling the robot safety from the initial state for a pre-determined number of time steps, with an additional clause representing the liveness condition being required to hold at the final time step for that depth. Consider the livelocked specification in Listing 6.

Listing 6 Core-finding example – unsatisfiable livelock

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

Unrolling the robot safety to depth d, with the added clause for liveness at depth d, results in:

$$\begin{aligned}\psi^d_{fromInit} \;=\; & \pi^1_{kitchen} \wedge \bigwedge_{1\le i\le d} \neg\pi^i_{hall\_w} \\ & \wedge \pi^d_{r3} \wedge \bigwedge_{2\le i\le d} \pi^i_{camera} \wedge \bigwedge_{1\le i\le d} \varphi^i_{topo}\end{aligned}$$

where φ^i_{topo} represents the topology constraints on the robot unrolled at time i. $\psi^d_{fromInit}$ is unsatisfiable for any $d \geq 1$.

In the case of deadlock, the propositional formula $\psi^d_{fromInit}$ can be built for increasingly larger depths until it is found to be unsatisfiable for some d; by the definition of deadlock, there will always exist such a d. This gives us a sound and complete method for determining the depth to which the safety formula must be unrolled in order to identify an unsatisfiable core for deadlock. For livelock, on the other hand, determining the shortest depth that will produce a meaningful core is a much bigger challenge. Consider the above example. For unroll depths less than or equal to 3, the unsatisfiable core re-

turned will include just the environment topology, since the robot cannot reach r3 from the kitchen in 3 steps or fewer, even if it is allowed into $hall\_w$; however, this is not a meaningful core.

For $d \geq 3$ the core is given by the subformula:

$$\pi^1_{kitchen} \wedge \bigwedge_{1 \leq i \leq d} \neg \pi^i_{hall\_w} \wedge \pi^d_{r3} \wedge \bigwedge_{1 \leq i \leq d} \varphi^i_{topo},$$

which maps back to specification sentences (1), (2) and (4). This is because the robot cannot reach $r3$ without passing through $hall\_w$. Section 5.5 contains another example demonstrating unsatisfiable core-finding for livelock.

The depth required to produce a meaningful core for unsatisfiability is bounded above by the number of distinct states that the robot can be in, i.e. the number of possible truth assignments to all the input and output propositions. However, efficiently determining the shortest depth that will produce a meaningful core remains a future research challenge, and for the purpose of this work, a fixed depth of 15 time steps was used for the examples presented, unless otherwise indicated.

5.2.3 Interactive Exploration of Unrealizable Tasks

If the specification is unrealizable rather than unsatisfiable, the above techniques do not apply directly to identify a core. This is because, if the specification is satisfiable but unrealizable, there exist sequences of truth assignments to the input variables that allow the system requirements to be met. Therefore, in order to produce an unsatisfiable Boolean formula, all sequences of truth assignments to the input variables that satisfy the environment assumptions must be considered. This requires one depth-d Boolean unrolling for each possible length-d sequence of inputs, where each unrolling encodes a

distinct sequence of inputs in the unrolled Boolean formula. In the worst case, the number of depth-d Boolean formulas that must be generated before an unsatisfiable formula is found grows exponentially in d. time step However, unsatisfiable cores do enable a useful enhancement to an interactive visualization of the environment counterstrategy. Since succinctly summarizing the cause of an unrealizable specification is often challenging even for humans, one approach to communicating this cause in a user-friendly manner is through an interactive game (described previously in Section 4.4 and depicted in Figure 5.3). The tool illustrates environment behaviors that will cause the robot to fail, by letting the user play as the robot against an adversarial environment. At each discrete time step, the user is presented with the current goal to pursue and the current state of the environment. They are then able to respond by changing the location of the robot and the status of its actuators. Examples of this tool in action are given in [48, 38].

The initial version of this tool (in Section 4.4) simply prevented the user from making moves that were disallowed by the specification. However, by using the above core-finding technique, a specific explanation can be given about the part of the original specification that would be violated by the attempted invalid move [46]. This is achieved by finding the unsatisfiable core of a single-step satisfiability problem constructed over the user's current state, desired next state, and the specified robot safety conditions.

For example, in the case of Listing 7, we discover that the robot cannot achieve its goal of following the user (Line 1) if the user enters the kitchen (which the robot has been banned from entering in Line 2). This conflict is presented to the user as follows: the environment sets its state to represent the target's being in the kitchen, and then, when the user attempts to enter the kitchen, the tool explains that this move is in conflict with Line 2.

Listing 7 Example of unrealizability

1. Follow me.
2. Avoid the kitchen.

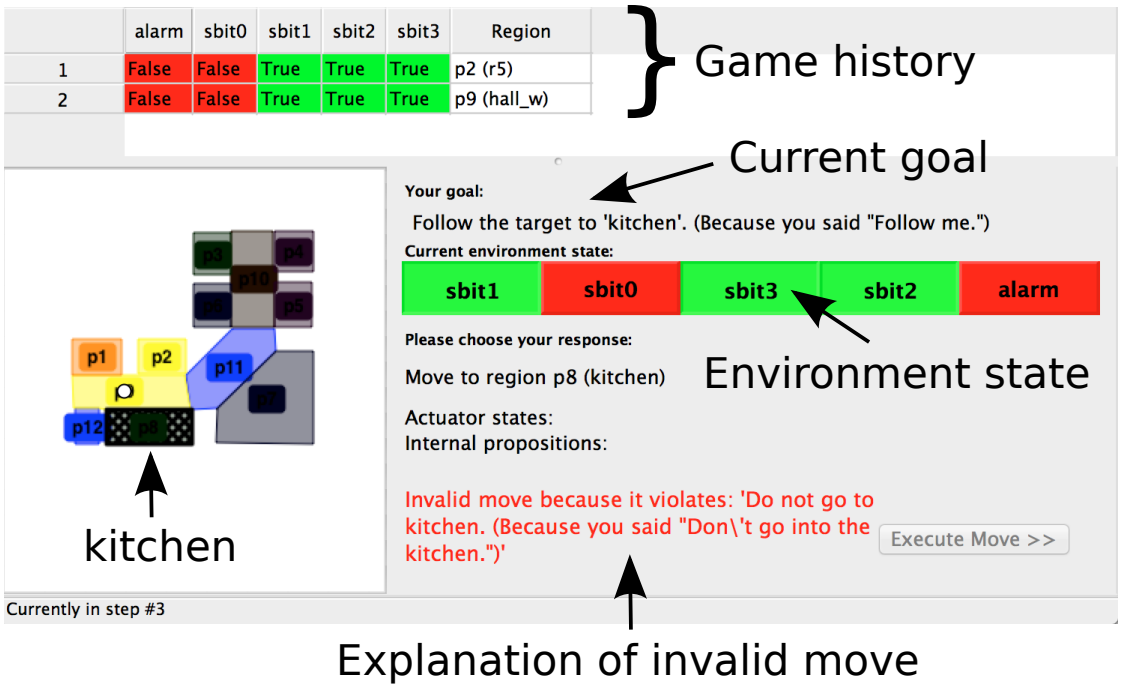


Figure 5.3: Screenshot of interactive visualization tool for the specification in Listing 7. The user is prevented from following the target into the kitchen in the next step (denoted by the blacked out region) due to the portion of the specification displayed.

5.3 Unrealizable Cores via Propositional SAT

As described in Section 5.2.3, the extension of the SAT-based core-finding techniques described in Section 5.2 to unrealizable specifications requires examining the exact environment input sequences that cause the failure. Considering all possible environment input sequences is not feasible; fortunately, the environment counterstrategy provides us with exactly those input sequences that cause unsynthesizability.

In this section, unrealizable components of the robot specification φ_s are analyzed based on the environment counterstrategy, narrowing down the cause of un-

realizability for both deadlock and livelock. Consider a counterstrategy $A^e_\varphi = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta_e, \delta_s, \gamma_\mathcal{X}, \gamma_\mathcal{Y}, \gamma_{goals})$ for formula φ. It allows the following characterizations of deadlock and livelock:

- **Deadlock** There exists a state in the counterstrategy such that there is a truth assignments to inputs, for which no truth assignment to outputs satisfies the robot transition relation. Formally,

$$\exists q \in Q \text{ s.t. } \delta_s(q, \delta_e(q)) = \emptyset$$

- **Livelock** There exist a set of states $\mathcal{C}$ in the counterstrategy such that the robot is trapped in $\mathcal{C}$ no matter what it does, and there is some robot liveness B_k in φ^g_s that is not satisfied by any state in $\mathcal{C}$. Formally,

$$\exists \mathcal{C} \subseteq Q, B_k \preceq \varphi^g_s \text{ s.t. } \forall q \in \mathcal{C}, q \not\models B_k, \delta_s(q, \delta_e(q)) \subseteq \mathcal{C}$$

5.3.1 Unrealizable Cores for Deadlock

Consider the specification in Listing 8, which refers to the workspace in Figure 5.1. The robot starts in $r5$ with the camera on (1). The safety conditions specify that the robot should not pass through the region marked $r5$ if it senses a person (2). In addition, the robot must stay in place if it senses a person (3). Finally, the robot should always activate its camera (4). Here, the environment can force the robot into deadlock by activating the "person" sensor (π_{person}) when the robot is in $r5$, because there is then no way the robot can fulfill both (2) and (3).

The environment counterstrategy A^e_φ is as follows:

- $Q = Q_0 = \{q_1\}$

Listing 8 Core-finding example – deadlock

1. Robot starts in r5 with camera (φ_s^i):
   $\pi_{r5} \wedge \pi_{camera}$
2. If you are sensing person then do not r5 (φ_s^t):
   $\square(\bigcirc \pi_{person} \Rightarrow \neg \bigcirc \pi_{r5})$
3. If you are sensing person then stay there (φ_s^t):
   $\square(\bigcirc \pi_{person} \Rightarrow (\bigcirc \pi_{start} \Leftrightarrow \pi_{start} \wedge \bigcirc \pi_{r2} \Leftrightarrow \pi_{r2} \ldots))$
4. Always activate your camera (φ_s^t):
   $\square \bigcirc \pi_{camera}$

- $\mathcal{X} = \{\pi_{person}\}$, $\mathcal{Y} = \{\pi_{r5}, \pi_{camera}\}$
- $\delta_e(q_1) = \{\pi_{person}\}$, $\delta_s(q_1, \{\pi_{person}\}) = \emptyset$, $\delta_s(q_1, \emptyset) = \emptyset$
- $\gamma_{\mathcal{X}}(q_1) = \{\pi_{person}\},\ \gamma_{\mathcal{Y}}(q_1) = \{\pi_{r5}, \pi_{camera}\}$
- $\gamma_{goals}(q_1) = 1$

The state q_1 is deadlocked, because given the input π_{person} in the next time step, there is a conflict between safety conditions 2 and 3, and the robot has no valid move (so $\delta_s(q_1, \{\pi_{person}\}) = \emptyset$). Note that $\delta_s(q_1, \emptyset) = \emptyset$ indicates that the environment strategy does not include any transition out of q_1 where the environment does not activate the "person" sensor.

For $q \in Q$, the *propositional-representation of q* is defined as:

$$\psi_{state}(q) = \bigwedge_{x \in \gamma_{\mathcal{X}}(q)} x \wedge \bigwedge_{x \in \mathcal{X} \setminus \gamma_{\mathcal{X}}(q)} \neg x \wedge \bigwedge_{y \in \gamma_{\mathcal{Y}}(q)} y \wedge \bigwedge_{y \in \mathcal{Y} \setminus \gamma_{\mathcal{Y}}(q)} \neg y$$

In the example above, $\psi_{state}(q_1) = \pi_{person} \wedge \pi_{r5} \wedge \pi_{camera}$.

Let π^i represent the value of $\pi \in AP$ at time step i, and $AP^i = \{\pi^i \mid \pi \in AP\}$. For example, in Listing 8, $AP^0 = \{\pi^0_{person}, \pi^0_{r5}, \pi^0_{camera}\}$ and $AP^1 = \{\pi^1_{person}, \pi^1_{r5}, \pi^1_{camera}\}$.

Given LTL specification φ, $q \in Q$ such that $\delta_s(q, \delta_e(q)) = \emptyset$, construct a propositional formula over $AP^0 \cup AP^1$ as follows:

$$\psi_{dead}(q, \varphi) = \psi_{state}(q)[\pi/\pi^0] \wedge \bigwedge_{z \in \delta_e(q)} z^1 \wedge \bigwedge_{z \in \mathcal{X} \setminus \delta_e(q)} \neg z^1$$
$$\wedge\, \varphi_s^t[\bigcirc \pi/\pi^1][\pi/\pi^0],$$

Intuitively, this formula represents the satisfaction of the robot safety condition in the next step from state q, with the additional restriction that the input variables be bound to the values provided by $\delta_e(q)$ in the next time step.

In the above case, $\psi_{dead}(q_1, \varphi) =$

$$\pi^0_{person} \wedge \pi^0_{r5} \wedge \pi^0_{camera} \wedge \pi^1_{person} \wedge \pi^1_{camera} \wedge \varphi^0_{topo}$$
$$\wedge(\pi^1_{person} \Rightarrow \neg\pi^1_{r5}) \wedge (\pi^1_{person} \Rightarrow (\pi^1_{r5} \Leftrightarrow \pi^0_{r5} \wedge ...)),$$

where φ^i_{topo} is a formula over $AP^i \cup AP^{i+1}$ representing the topological constraints on the robot motion at time i (i.e. which rooms it can move to at time $i+1$ given where it is at time i, and mutual exclusion between rooms).

Note that if q is a deadlocked state, then by definition $\psi_{dead}(q, \varphi)$ is unsatisfiable, since there is no valid setting to the robot propositions in the next time step starting from q. A SAT solver can now be used to find a minimal unsatisfiable subformula, as in Section 5.2.

In the above example, the SAT solver finds the core of $\psi_{dead}(q, \varphi)$ as the subformula

$$\pi^0_{r5} \wedge \pi^1_{person} \wedge (\pi^1_{person} \Rightarrow \neg\pi^1_{r5}) \wedge (\pi^1_{person} \Rightarrow (\pi^1_{r5} \Leftrightarrow \pi^0_{r5})).$$

This is because the two statements combined require the robot to both stay in r5 and not be in r5 in time step 1. This gives us a core explanation of the deadlock caused in state q. Taking the union over the cores for all the deadlocked states provides a concise explanation of how the environment can force the robot into a deadlock situation. Section

6.5 contains another, more complex example demonstrating unrealizable core-finding for deadlocked specifications.

5.3.2 Unrealizable Cores for Livelock

Consider Listing 4 again. If the environment action is to always set π_{person}, then the safety requirement in 2 enforces that the robot will never activate π_{r5}, because it is explicitly forbidden from doing so when sensing a person. This is livelock because the robot can continue to move between $start$ and $r2-r4$. The environment counterstrategy A^e_φ is as follows:

- $Q = \{q_1, q_2, q_3, q_4\},\ Q_0 = \{q_1\}$
- $\mathcal{X} = \{\pi_{person}\}, \mathcal{Y} = \{\pi_{start}, \pi_{r2}, \pi_{r3}, ..., \pi_{r8}, \pi_{goal}, \pi_{camera}\}$
- $\forall q \in Q, \delta_e(q) = \{\pi_{person}\}$
- $$\delta_s(q_i, \{\pi_{person}\}) = \begin{cases} \{q_1, q_2\} & \text{if } i = 1 \\ \{q_{i-1}, q_i, q_{i+1}\} & \text{for } 2 \leq i \leq 3 \\ \{q_3, q_4\} & \text{if } i = 4 \end{cases}$$

  Additionally, $\delta_s(q, \emptyset) = \emptyset$ for all q, and $\delta_s(q, *) = \emptyset$ for $q \in \{q_5, .., q_8, q_{goal}\}$
- $\forall q \in Q, \gamma_{\mathcal{X}}(q) = \{\pi_{person}\}$.
- $$\gamma_{\mathcal{Y}}(q_i) = \begin{cases} \{\pi_{camera}, \pi_{start}\} & \text{if } i = 1 \\ \{\pi_{camera}, \pi_{r_i}\} & \text{for } 2 \leq i \leq 4 \end{cases}$$
- $\forall i, \gamma_{goals}(q_i) = 1$ (since there is only one goal).

In the case of livelock, we know there exists a set of states $\mathcal{C}$ in the counterstrategy that trap the robot, locking it away from the goal. Without loss of generality, $\mathcal{C}$ consists of cycles of states. In the specifications of the form considered in this work, robot goals are of the form $\square\diamondsuit B_i$ for $1 \leq i \leq n$, where each B_i is a propositional formula over $AP = \mathcal{X} \cup \mathcal{Y}$. Suppose the algorithm in [38] identified goal $\square\diamondsuit B_k$ as the goal responsible for livelock. Let Q_k be the set of all states in A^e_φ that prevent goal B_k, and let $\mathcal{C}_k$ be the set of *maximal k-preventing cycles* in Q_k, i.e. cycles that are not contained in any other cycle in Q_k (modulo state-repetition). Let $C_1 = (q^1_0, q^1_1, ..., q^1_a)$ and $C_2 = (q^2_0, q^2_1, ..., q^2_b)$ be cycles, and define $C_1 \prec C_2$ if $a < b$ and there is some offset index o in C_2 such that all of C_1 is found in C_2 starting at o, i.e. $q^1_i = q^2_{(i+o) \bmod (b+1)}$ for all $0 \leq i \leq a$. This expresses that C_1 is a strict sub-cycle of C_2. Formally,

$$
\begin{aligned}
Q_k &= \{q \in Q | \gamma_{goals}(q) = k\} \\
\\
\mathcal{C}^{all}_k &= \{(q_0, q_1, ..., q_l) | \forall 0 \leq i \leq l, q_i \in Q_k, \\
&\qquad \forall i < l, q_{i+1} \in \delta_s(q_i, \delta_e(q_i)), q_i \neq q_{i+1}, \\
&\qquad q_0 \in \delta_s(q_l, \delta_e(q_l)), q_0 \neq q_l\}, \\
\\
\mathcal{C}_k &= \{C \in C^{all}_k | \forall C' \in C^{all}_k, C \not\prec C'\}.
\end{aligned}
$$

In the specification in Listing 4, there is only one goal, $\square\diamondsuit \pi_{goal}$. $C_1 = (q_1, q_2, q_3, q_4, q_3, q_2)$ is a maximal 1-preventing cycle.

Given an initial state q, a depth d and an LTL safety formula φ^t_s over $\pi \in AP$, there exists a propositional formula $\psi^d(\varphi^t_s, q)$ over $\bigcup_{0 \leq i \leq d+1} AP^i$, constructed as:

$$
\psi^d(\varphi^t_s, q) = \psi_{state}(q)[\pi/\pi^0] \wedge \bigwedge_{0 \leq i \leq d} \varphi^t_s[\bigcirc \pi/\pi^{i+1}][\pi/\pi^i].
$$

This formula is called the *depth-d unrolling of φ^t_s from q*, and represents the tree of length-$d+1$ truth assignment sequences that satisfy φ^t_s, starting from q. Note that there

are $d+1$ time steps in a depth-d unrolling because each conjunct in φ_s^t governs two time steps. In the example, $\psi^d(\varphi_s^t, q_1) =$

$$\pi^0_{start} \wedge \pi^0_{camera} \wedge \bigwedge_{0 \leq i \leq d} (\varphi^i_{topo} \wedge \pi^{i+1}_{camera} \wedge (\pi^{i+1}_{person} \Rightarrow \neg \pi^{i+1}_{r5})).$$

Given a cycle of states $C = (q_0, q_1, ..., q_l) \in A^e_\varphi$, and a depth d, construct a propositional formula $\psi^d_{\mathcal{X}}(C)$ over $\bigcup_{0 \leq i \leq d} \mathcal{X}^i$, where $x^i \in \mathcal{X}^i$ represents the value of each input $x \in \mathcal{X}$ in state $q_{i \bmod (l+1)}$ for $0 \leq i \leq d$, as:

$$\psi^d_{\mathcal{X}}(C) = \bigwedge_{0 \leq i \leq d} \Big(\bigwedge_{p \in \gamma_{\mathcal{X}}(q_{i \bmod (l+1)})} x^i \wedge \bigwedge_{p \in \mathcal{X} \setminus \gamma_{\mathcal{X}}(q_{i \bmod (l+1)})} \neg x^i \Big).$$

This formula is called the *depth-d environment-unrolling of C*, and represents the sequence of inputs seen when following cycle C for d time-steps. In the example, the depth-d environment unrolling of C_1 is $\psi^d_{\mathcal{X}}(C) = \bigwedge_{0 \leq i \leq d} \pi^i_{person}$.

Now, given an LTL safety specification φ_s^t over $\pi \in AP$, a goal B_k, a maximal k-preventing cycle $C_k = (q_0, q_1, ..., q_l) \in \mathcal{C}_k$, and an unrolling depth d, construct propositional formula $\psi^d_{live}(B_k, C_k, \varphi_s^t)$ over $\bigcup_{0 \leq i \leq d+1} AP^i$ as:

$$\psi^d_{live}(B_k, C_k, \varphi_s^t) = \\ \psi^{d+1}_{\mathcal{X}}(C_k) \wedge \psi^d(\varphi_s^t, q_0) \wedge B_k[\pi / \pi^d].$$

Intuitively, this formula expresses the requirement that the goal B_k be fulfilled after some depth-d unrolling of the safety formula starting from state q_0, given the input sequence provided by $\psi^{d+1}_{\mathcal{X}}(C_k)$ (note that this input sequence extends to the final time step in the safety formula unrolling). Again, this is an unsatisfiable propositional formula, and can be used to determine the core set of clauses that prevent a goal from being fulfilled. Taking the union of cores over all $C_k \in \mathcal{C}_k$ gives a concise explanation of the ways in which the environment can prevent the robot from fulfilling the goal.

In the above example, $\psi^d_{live}(\pi_{goal}, C_1, \varphi^t_s) =$

$$\pi^0_{start} \wedge \pi^0_{camera} \wedge \bigwedge_{0 \leq i \leq d+1} \pi^i_{person}$$
$$\wedge \bigwedge_{0 \leq i \leq d} (\varphi^i_{topo} \wedge \pi^{i+1}_{camera} \wedge (\pi^{i+1}_{person} \Rightarrow \neg \pi^{i+1}_{r5}))$$
$$\wedge \pi^d_{goal}.$$

In the case of livelock, the choice of unroll depth d determines the quality of the core returned. Recall that for deadlock, the propositional formula $\psi_{dead}(q, \varphi)$ is built over just one step, since it is already known to cause a conflict with the robot transition relation, and be unsatisfiable. The unsatisfiable core of this formula is a meaningful unrealizable core in this case because it provides the immediate reason for the deadlock. For livelock, on the other hand, determining the shortest depth to which a cycle C_k must be unrolled to produce a meaningful core is not obvious.

In the above example, for unroll depths less than or equal to 8, the unsatisfiable core returned will include just the environment topology, since the robot cannot reach the goal from the start in 8 steps or fewer, even if it is allowed into $r5$; however, this is not a meaningful core. Unrolling to depth 9 or greater returns the expected subformula that includes $\bigwedge_{0 \leq i \leq d}(\pi^{i+1}_{person} \Rightarrow \neg \pi^{i+1}_{r5})$. Automatically determining the shortest depth that will produce a meaningful core remains a research challenge, but a good heuristic is to use the maximum distance between two states in the environment counterstrategy (i.e. the diameter of the graph representing the counterstrategy, or the sum of the diameters of its connected components). For the examples presented in this work, a depth of 15 time steps was used.

Note that, since unsatisfiability is a special case of unrealizability (in which not just *some*, but *any* environment can prevent the robot from fulfilling its specification), the above analysis also applies to unsatisfiable specifications. However, the analysis presented in Section 5.2 is more efficient for unsatisfiability, as it does not require explicit-

state extraction of the environment counterstrategy.

5.4 Unsynthesizable Cores via Iterated Synthesis

As discussed in Sections 5.2 and 5.3, the SAT-based approach to identifying an unsynthesizable core for the case of livelock presents the challenge of determining a depth to which the LTL formula must be instantiated with propositions. This minimal depth is often tied to the number of regions in the robot workspace, and is usually easy to estimate. However, no efficient, sound method is known for determining this minimal unrolling depth. Therefore, the SAT-based analysis in Section 5.3 may for some examples return a core that does not capture the real cause of failure. If this is observed and flagged by the user, alternative, more computationally expensive techniques can be used to return a minimal core.

This section presents one such alternative approach to determining the minimal subset of the robot safety conditions that conflicts with a specified goal. The approach is based on iterated realizability checks, removing conjuncts from the safety formula and testing realizability of the remaining specification. While this approach is guaranteed to yield a minimal unsynthesizable core, it requires repeated calls to a realizability oracle, which may be expensive for specifications with a large number of conjuncts.

Recall from Chapter 3 the syntactic form of the LTL specifications considered in this work. In particular, the formula φ_s^g is a conjunction $\bigwedge_{j=1}^{n_s^g} \square \diamondsuit B_j$, where each B_j is a Boolean formula over AP, and represents an event that should occur infinitely often when the robot controller is executed. Similarly, φ_s^t represents the robot safety constraints; it is a conjunction $\bigwedge_{i=1}^{n_s^t} \square A_i$ where each A_i is a Boolean formula over AP and $\bigcirc AP$.

In the case of livelock, the initial specification analysis presented in [38] provides a specific liveness condition B_k that causes the unsynthesizability (i.e. either unsatisfiability or unrealizability), and can also identify one of the initial states $\varphi_s^{badInit}$ from which the robot cannot fulfill B_k. However, the specific conjuncts of the safety formula φ_s^t that prevent this liveness are not identified. The key idea behind using realizability tests to determine an unrealizable or unsatisfiable core of safety formulas is as follows. If on removing a safety conjunct from the robot formula, the specification remains unsynthesizable, then there exists an unsynthesizable core that does not include that conjunct (since the remaining conjuncts are sufficient for unsynthesizability). Therefore, in order to identify an unsynthesizable core, it is sufficient to iterate through the conjuncts of φ_s^t, removing safety conditions one at a time and checking for realizability.

Algorithm 7 presents the formal procedure for performing these iterated tests, given the index k of the liveness condition that causes the unsynthesizability. Denote by $\varphi_s[S, \varphi_s^{badInit}, k] \subseteq \varphi_s$ the formula $\varphi_s^{badInit} \wedge \bigwedge_{i \in S} \square A_i \wedge \square \diamondsuit B_k$ for indices in a set S. Let S_i denote set S at iteration i. Set S_1 is initialized to the indices of all safety conjuncts, i.e. $S_1 = \{1, ..., n_s^t\}$ in line 2. In each iteration of the loop in lines 3-7, the next conjunct A_i is omitted from the robot transition relation, and realizability of $\varphi_e \Rightarrow \varphi_s[S_i \backslash \{i\}, \varphi_s^{badInit}, k]$ is checked (line 4). If removing conjunct i causes an otherwise unsynthesizable specification to become synthesizable, it is retained for the next iteration (line 5); otherwise it is permanently deleted from the set of conjuncts S_i (line 6-7). After iterating through all the conjuncts in $\{1, ..., n_s^t\}$, the final set $S_{n_s^t+1}$ determines a minimal unsynthesizable core of $\varphi_e \Rightarrow \varphi_s$ that prevents liveness k. Note that the core is non-unique, and depends both on the order of iteration on the safety conjuncts, and on the initial state $\varphi_s^{badInit}$ returned by the synthesis algorithm.

Theorem 5.4.1. *Algorithm 7 yields a minimal unsynthesizable core of* $\varphi_e \Rightarrow \varphi_s$.

Proof: Each iteration i of the loop in Algorithm 7, lines 3-7, maintains the invariant that $\varphi_e \Rightarrow \varphi_s[S_i, \varphi_s^{badInit}, k]$ is unsynthesizable; thus, $\varphi_e \Rightarrow \varphi_s[S_{n_s^t+1}, \varphi_s^{badInit}, k]$ is unsynthesizable when the loop is exited.

Moreover, removing any of the safety conjuncts in $S_{n_s^t+1}$ yields a synthesizable specification. To see this, assume for a contradiction that there exists $j \in S_{n_s^t+1}$ such that $\varphi_e \Rightarrow \varphi_s[S_{n_s^t+1} \backslash \{j\}, \varphi_s^{badInit}, k]$ is unsynthesizable. Clearly, $S_{n_s^t+1} \subseteq S_j$, so by definition of $\preceq$, $\varphi_s[S_{n_s^t+1} \backslash \{j\}, \varphi_s^{badInit}, k] \preceq \varphi_s[S_j \backslash \{j\}, \varphi_s^{badInit}, k]$. Therefore, if $\varphi_e \Rightarrow \varphi_s[S_j \backslash \{j\}, \varphi_s^{badInit}, k]$ is synthesizable, then $\varphi_e \Rightarrow \varphi_s[S_{n_s^t+1} \backslash \{j\}, \varphi_s^{badInit}, k]$ must be synthesizable, since any implementation that satisfies $\varphi_s[S_j \backslash \{j\}, \varphi_s^{badInit}, k]$ also satisfies $\varphi_s[S_{n_s^t+1} \backslash \{j\}, \varphi_s^{badInit}, k]$. Since j was not removed from S_j on the j^{th} iteration, $\varphi_e \Rightarrow \varphi_s[S_j \backslash \{j\}, \varphi_s^{badInit}, k]$ is synthesizable. It follows that $\varphi_e \Rightarrow \varphi_s[S_{n_s^t+1} \backslash \{j\}, \varphi_s^{badInit}, k]$ must be synthesizable, a contradiction.

Algorithm 7 Unsynthesizable Cores via Iterated Realizability Testing

1: **INPUT:** $\varphi_e, \varphi_s, \varphi_s^{badInit}, k$
2: $S_1 = \{1, 2, ..., n_s^t\}$
3: **for** $i := 1$ to n_s^t **do**
4: **if** $(\varphi_e \Rightarrow \varphi_s[S_i \backslash \{i\}, \varphi_s^{badInit}, k])$ is synthesizable **then**
5: $S_{i+1} \leftarrow S_i$
6: **else**
7: $S_{i+1} \leftarrow S_i \backslash \{i\}$
8: **OUTPUT:** $\varphi_s[S_{n_s^t+1}, \varphi_s^{badInit}, k])$

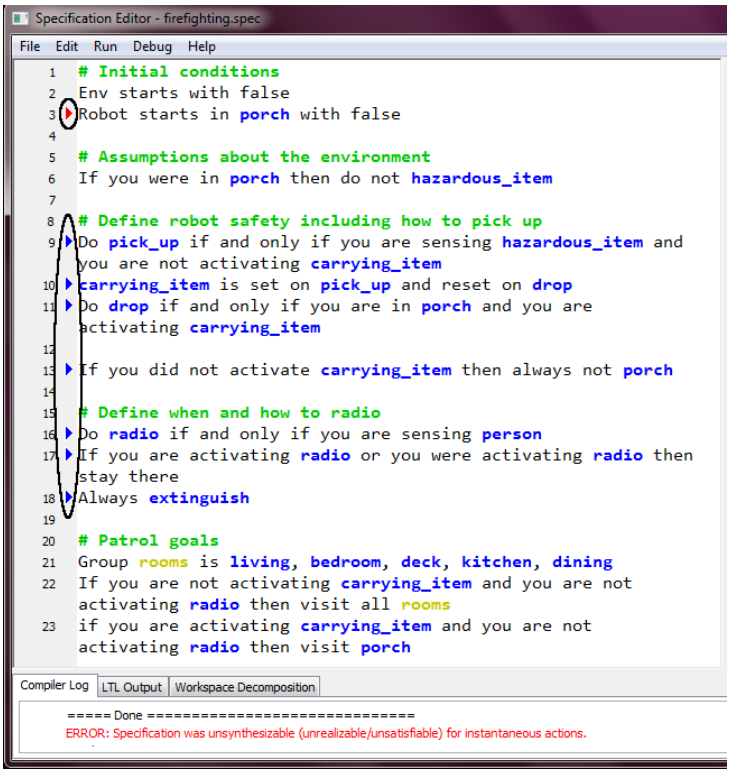
Note that Algorithm 7 yields an unsynthesizable core for livelock, for both unsatisfiable and unrealizable specifications. It is sound and complete, because it will always yield a minimal set of safety conditions that prevent the relevant liveness. As compared with the methods presented in Section 5.2 and 5.3, it circumvents the problem of determining the depth to which to instantiate the LTL safety formula in a propositional SAT instance. Moreover, if $\varphi_s[S, \varphi_s^{badInit}, k]$ is replaced with $\varphi_s^{badInit} \wedge \bigwedge_{i \in S} \Box A_i \wedge \Box \Diamond$ TRUE (i.e. the robot liveness condition is trivial), the algorithm also yields an

unsynthesizable core in the case of deadlock.

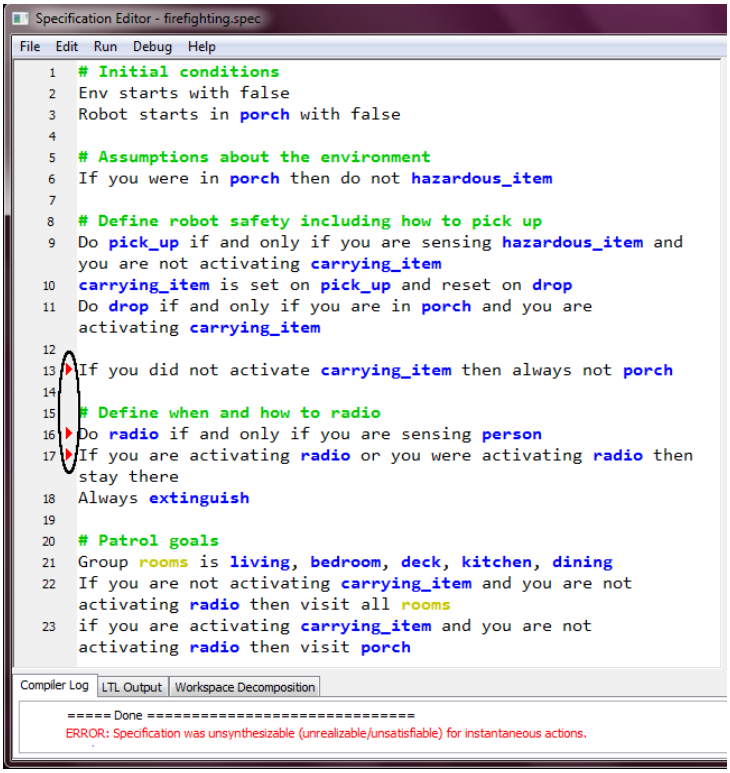
However, there is a computational tradeoff involved in performing a synthesizability (i.e. realizability) check once for every conjunct in the safety formula, instead of once for the entire specification. Algorithm 7 checks synthesizability once in each iteration of the loop in lines 3-7. Using the efficient algorithm in [37], each realizability check takes time $O((mn\Sigma)^3)$, where Σ is the size of the state space, i.e. $\Sigma = 2^{|\mathcal{X}\cup\mathcal{Y}|}$, and m, n are the number of environment and system liveness conditions, respectively. Therefore the complexity of Algorithm 7 is $O((n_s^t)(mn\Sigma)^3)$. On the other hand, the complexity of the approach in Section 5.3 requires only one call to the counterstrategy synthesis algorithm, but multiple calls to the SAT solver. The SAT solver is invoked with Boolean formulas in CNF form that are, in the worst case, exponential in the size of the original LTL conjuncts. However, iterated realizability tests do not require explicit extraction of the environment counterstrategy, unlike the SAT-based tests presented in Section 5.3. The relative appropriateness of the two methods (SAT-based vs. iterated realizability testing) for the case of deadlock will depend on the specific unsynthesizable formula.

5.5 Examples

This section presents examples of the cores identified for unsynthesizable specifications. The examples presented previously appeared in [48], and this section demonstrates the improvement of the proposed approach over the analysis presented in that work.

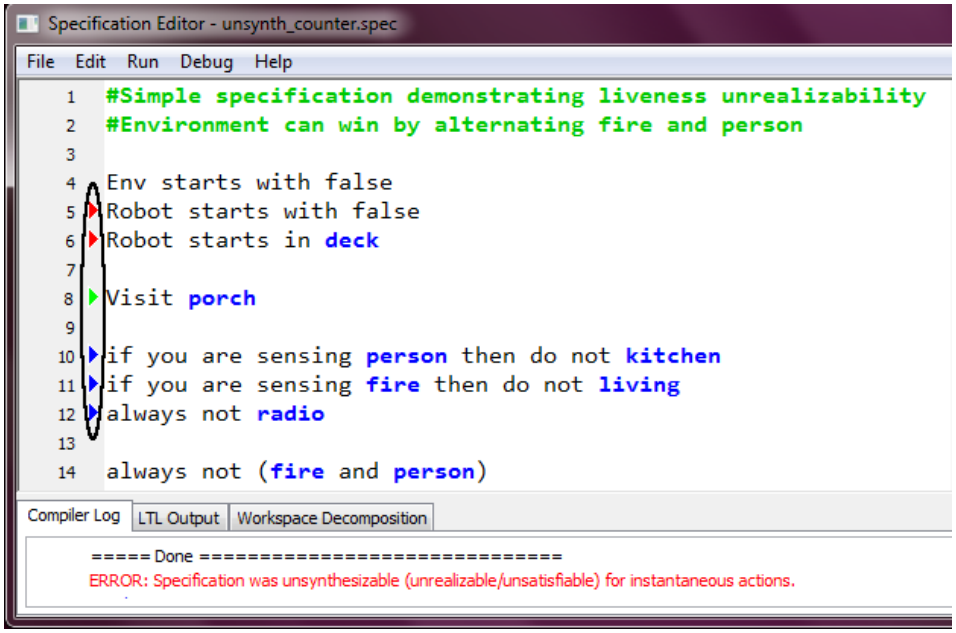


(a) Sentences highlighted using approach in Chapter 4

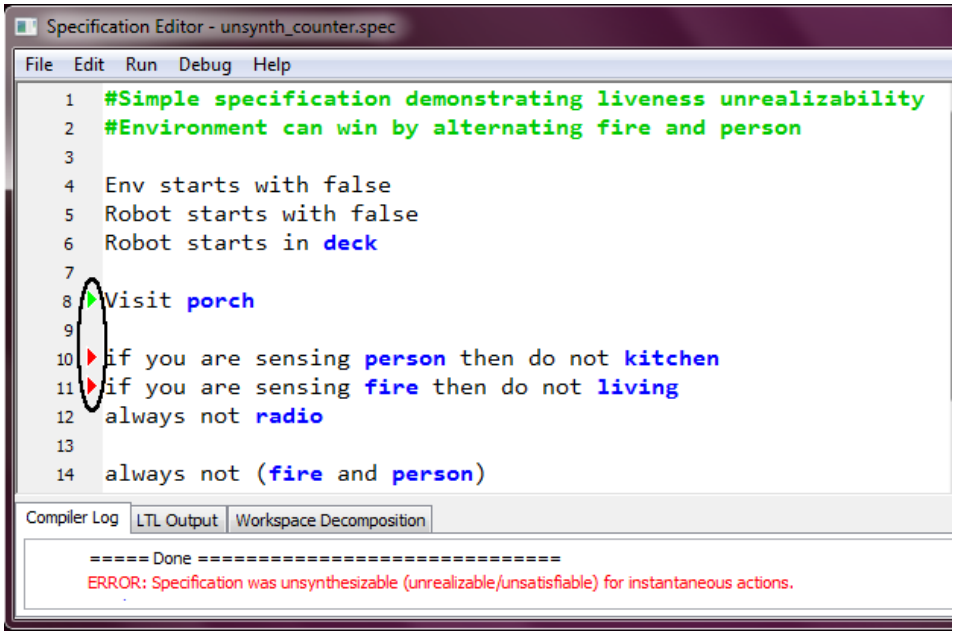


(b) Sentences highlighted using approach in Chapter 5

Figure 5.4: Core-Finding Example: Deadlock



(a) Sentences highlighted using approach in Chapter 4



(b) Sentences highlighted using approach in Chapter 5

Figure 5.5: Core-Finding Example: Livelock

5.5.1 Deadlock

Consider the specification in Figure 5.4, where the robot is operating in the workspace depicted in 3.1(b). The robot starts in the porch. The safety conditions govern what it should do when it senses a "person" (stay with them and radio for help) or a "hazardous item" (pick up the hazardous item and carry it to the porch). The robot should not return to the porch unless it is carrying a hazardous item. The robot's goals are to patrol all rooms in the workspace.

The environment can cause deadlock by setting the person sensor to true and the hazardous item sensor to false when the robot is in the porch. Note that sensing a hazardous item results in the robot activating the "pick_up" action, which in turn results in the proposition "carrying_item" being set. Similarly, sensing a person results in the robot turning on the radio. Now the state in which both "radio" and "carrying_item" are true in the porch is deadlocked because of the safety conditions, "If you are activating radio or you were activating radio then stay there" and "If you did not activate carrying_item then always not porch", since there is no way to satisfy both from this state.

Figure 5.4(a) depicts the sentences highlighted by the algorithm described in Chapter 4. A subset of sentences in the specification is identified by triangle-shaped markers in the left-hand margin, and the color-coding is based on whether they correspond to initial, safety or liveness conditions. The sentences highlighted in 5.4(a) include all initial (red) and safety (blue) conditions, which forms a very large subset of the original specification. On the other hand, Figure 5.4(b) depicts the much smaller subset of guilty sentences returned by the analysis presented in this chapter (these sentences are all highlighted in red). The core sentences highlighted correspond to the safety conditions that cause deadlock – in this example, removing any one of these sentences results in a synthesizable specification.

5.5.2 Livelock

Consider the specification in Figure 5.5, also in the same workspace. The robot starts in the deck and its goal is to visit the porch. However, based on whether it senses a person or a fire, it has to keep out of the kitchen and the living room, respectively. Figure 5.5(a) depicts the sentences highlighted by the algorithm in Chapter 4, which includes all safety conditions (red) in addition to the goal (green). This includes irrelevant sentences, such as the one that requires the robot to always turn on the camera. Figure 5.5(b) depicts the core returned by the analysis in this work – only those safeties that directly contribute to keeping the robot out of the porch are returned.

5.6 Conclusions

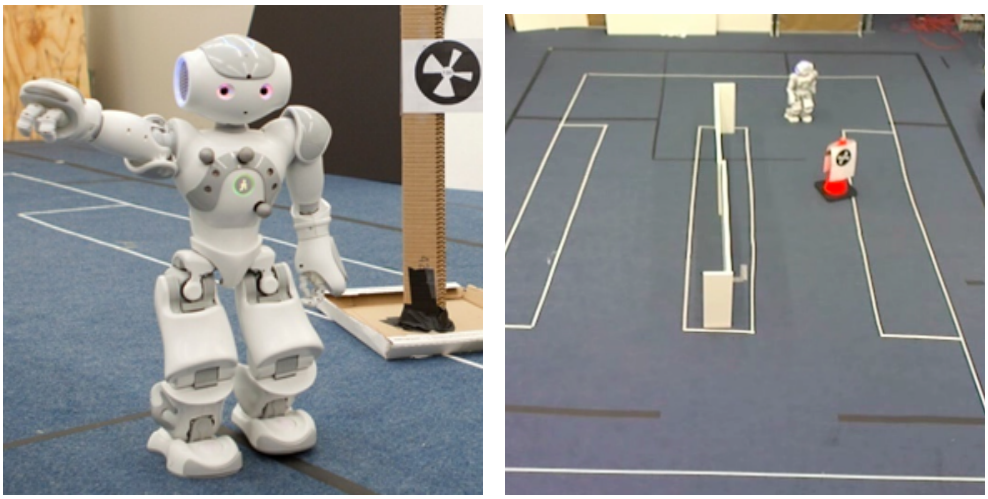
This chapter provides techniques for analyzing high-level robot specifications that are unsynthesizable, with the goal of providing a minimal explanation for why the robot specification is inconsistent, or how the environment can prevent the robot from fulfilling the desired guarantees. The causes of failure presented in this work take the form of unsynthesizable subsets of the original specification, or *cores*. A suite of SAT-based techniques is presented for identifying unsatisfiable and unrealizable cores in the case of deadlock and most cases of livelock; iterated realizability checking is used to identify cores in cases where the SAT-based analysis fails. Examples show that the additional analysis provides improvements in terms of reducing the number of flagged sentences in the original specification, and ignoring irrelevant subformulas.

Future work includes automatically determining the depth for obtaining a meaningful core in the case of livelock for the SAT-based approaches, and exploring SAT-based

techniques that do not require explicit state extraction of the counterstrategy automaton. Another direction for future study is the empirical comparison of SAT-based techniques with approaches based on iterated realizability testing, to evaluate relative computation time for practical examples.

CHAPTER 6

TIMING SEMANTICS FOR ABSTRACTION AND EXECUTION OF SYNTHESIZED HIGH-LEVEL ROBOT CONTROL



(a) Aldebaran Nao close-up (b) Nao in the workspace for Example 5

Figure 6.1: An experiment with the Aldebaran Nao that demonstrates the problem of actions with different execution durations.

Robotics has recently seen the successful application of formal methods to the construction of controllers for high-level autonomous behaviors, including reactive conditions and infinite goals [27, 35, 7, 26, 43]. One technique that has been successfully applied to high-level robot planning is Linear Temporal Logic (LTL) synthesis, in which a correct-by-construction controller is automatically synthesized from a formal task specification[26, 43]. Synthesis-based approaches operate on a discrete abstraction of the robot workspace and a formal specification of the environment assumptions and desired robot behavior in LTL. Synthesis algorithms automatically construct an automaton guaranteed to fulfill the specification on the discrete abstraction (if such an automaton exists). The automaton is then used to create a hybrid controller that calls low-level continuous controllers corresponding to each discrete transition. During the execution of this hybrid controller, a single transition between discrete states in the automaton may correspond to the simultaneous execution of several low-level controllers.

Example 5. Consider an Aldebaran Nao robot, shown in Figure 6.1, performing a task in the lab [21]. The available actions for this robot include motion of the arm

(waving), a text-to-speech interface, and walking; walking between regions of interest takes significantly longer than any of the other actions.

In the discrete abstraction of the above problem, the robot's state encodes the robot's current location and whether it is waving. Suppose the implementing automaton contains a discrete transition from the state where the robot location is region r_1 and it is not waving, to the state where the robot location is r_2 and it is waving. This discrete transition corresponds to executing two continuous controllers – one for motion and one for waving; the controller for waving takes less time to complete execution than the motion between rooms.

In general, a robot with multiple action capabilities will use low-level controllers that take varying amounts of time to complete. When reasoning about correctness of continuous execution, most approaches make assumptions about the physical execution of actions given a discrete implementation, such as when actions will complete relative to each other, and possible changes in the robot's environment while it is performing various actions. Relaxing these assumptions gives rise to a number of challenges in the continuous implementation of automatically-synthesized hybrid controllers.

This chapter presents several approaches to discrete synthesis and continuous execution, and compares the assumptions they make on the robot's physical capabilities and the environment in which it operates. Assumptions on robot actions range in strength from instantaneous actuation to arbitrary but finite relative execution times. The approaches are also compared based on responsiveness to events in the environment, and assumptions made about when changes in the environment can occur. The framework handles a class of specifications corresponding to the Generalized Reactivity (1) (GR(1)) [37] fragment of Linear Temporal Logic, which captures a large number of high-level tasks specified in practice.

This is one of the first works to consider the safety and correctness of continuous executions of synthesized controllers arising from the physical nature of the problem domain. There are a few previous works that incorporate the continuous nature of the physical execution during the discrete synthesis process. For example, the authors of [22, 20] evaluate discrete controllers on optimality with respect to a continuous metric based on the physical workspace, and extract more optimal solutions at synthesis time. The problem of synthesizing provably correct continuous control has recently been addressed [45, 50]. The contents of this chapter supersede the work described in those works, and compares the two approaches. Further, it includes details of the modified synthesis algorithm that enables efficient synthesis for the approach in [50].

The remainder of this chapter is structured as follows. Section 6.1 presents the continuous controller execution paradigm introduced in [26], and the assumptions associated therewith. Section 6.2 presents the alternative paradigm and synthesis algorithm introduced in [45], in which actions are assumed to fall into two classes based on the duration of execution. Section 6.3 evaluates the two approaches on the assumptions they make and the behaviors they produce, and provides a formal problem statement aiming to relax these assumptions. Section 6.4 describes a controller-synthesis framework that produces controllers with provably correct continuous execution for arbitrary relative action execution times; this includes modifications to the synthesis algorithm that keep it tractable. Section 6.5 presents examples comparing the effectiveness of the three approaches. Section 6.6 discusses the challenge of providing user feedback on specifications that have no implementing controller because of the timing semantics in continuous execution. The chapter concludes with a description of future work in Section 6.7.

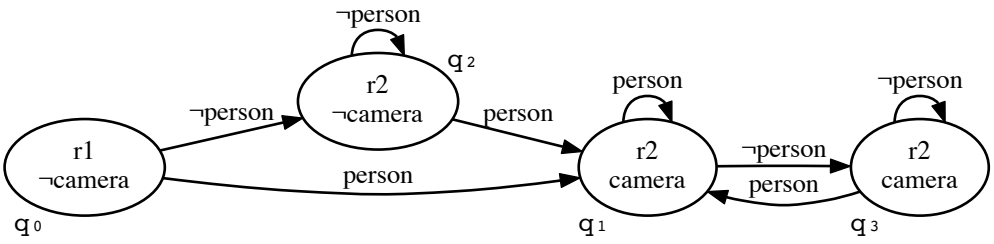


Figure 6.2: Synthesized automaton for Example 6. Each state is labeled with the truth assignment to location and action propositions in that state. Each transition is labeled with the truth assignment to sensor propositions that enables it.

6.1 Assuming Instantaneous Actions – Synchronous Action Completion [26]

Example 6, adapted from [45], will serve to demonstrate the continuous controller execution paradigm on a simple high-level task.

Example 6. Consider a simple two-room workspace where the two adjacent locations are labeled r_1 and r_2 (corresponding to π_{r_1} and π_{r_2}). The robot has one sensor, which senses a person (represented by proposition π_{person}), and one action, which is to turn a camera on or off (π_{camera}). The robot starts in room r_1 with the camera off. When it senses a person, it must turn on the camera. Once the camera is on, it must stay on. Finally, the robot must visit room r_2 infinitely often. Here $\mathcal{X} = \{\pi_{person}\}$ and $\mathcal{Y} = \{\pi_{r_1}, \pi_{r_2}, \pi_{camera}\}$; $\mathcal{L} = \{\pi_{r_1}, \pi_{r_2}\}$.

Note that the camera action is a simpler case, since it is modeled as having two binary states (on/off), and thus captured by a single Boolean proposition, assuming controllers for toggling its state.

The adjacency relation for Example 6 is

$$\varphi_{trans} = \ \square(\varphi_{r_1} \Rightarrow \bigcirc \varphi_{r_1} \vee \bigcirc \varphi_{r_2}) \wedge \square(\varphi_{r_2} \Rightarrow \bigcirc \varphi_{r_2} \vee \bigcirc \varphi_{r_1}).$$

In addition, the task in Example 6 corresponds to the following LTL specification:

$$(\varphi_{r_1} \wedge \neg\pi_{camera}) \qquad \text{\#Initial}$$

(Robot starts in region r1 with the camera off)

$$\wedge \quad \square(\bigcirc \pi_{person} \Rightarrow \bigcirc \pi_{camera}) \qquad \text{\#Safety}$$

(Activate the camera if you see a person)

$$\wedge \quad \square((\pi_{camera} \Rightarrow \bigcirc \pi_{camera})) \qquad \text{\#Safety}$$

(Camera stays on once turned on)

$$\wedge \quad \square \diamondsuit(\pi_{r_2}) \qquad \text{\#Liveness}$$

(Go to r2 infinitely often)

Figure 6.2 depicts the automaton synthesized for the above specification using this synthesis algorithm. Each state of the automaton is labeled with the truth assignment to location and action propositions in that state, and each transition is labeled with the truth assignment to sensor propositions required for that transition to be enabled. Incoming transitions therefore also determine the truth value of the sensor propositions for each state. The labels $\texttt{r}_\texttt{i}$, `camera` and `person` represent π_{r_i}, π_{camera} and π_{person} respectively.

Suppose the robot starts in room r_1, with its camera turned off and no person sensed (so it is in the initial state q_0 in Figure 6.2). Suppose it then senses a person. The safety condition $\square(\bigcirc \pi_{person} \Rightarrow \bigcirc \pi_{camera})$ requires it to turn on the camera. In order to fulfill its patrol goal, it will also try to go to room r_2. So the discrete transition in the automaton generated by the synthesis algorithm in [37] will be to state q_1. To execute the transition (q_0, q_1) at the continuous level, a motion controller and a controller for turning on the camera must both be invoked.

The controller synthesis framework presented in Chapter 3 assumes that all robot actions are instantaneous. This is usually a reasonable assumption to make when the robot controllers do not take a lot of time to complete. This includes tasks that do not involve slow tasks like motion. This assumption is often violated in practice without consequence when using the controllers synthesized. Under the continuous execution paradigm described in [26], all controllers except motion between adjacent regions are assumed to have instantaneous execution. Given a discrete transition between two states with different locations, the motion controller for driving the robot between regions is activated first, and the remaining controllers only activated (or deactivated) once the robot has crossed into the new region; all controllers thus complete synchronously. Thus, to execute the transition (q_0, q_1) depicted in Figure 6.2, the hybrid controller first activates the controller for moving from r_1 to r_2, and only once that boundary has been crossed will it activate the (instantaneous) controller for turning on the camera, completing the discrete transition (q_0, q_1). Figure 6.3(a) depicts the change in state for the transition (q_0, q_1), and how it corresponds to the progress of the continuous controllers.

6.2 Assuming Slow and Fast Action Classes – Synchronous Controller Activation [45]

While there are tasks for which the assumption of instantaneous actions is reasonable, there are a wide variety of tasks that beg a more sophisticated model of the timing semantics of actuation. This includes, for example, all tasks that involve robot motion. In an attempt to relax the assumption of instantaneous actions, the robot's actions are grouped into two sets based on controller execution duration [45]. This section summarizes the assumptions and controller synthesis framework proposed in that work.

6.2.1 Problem Statement

Assume that there are two kinds of low-level controllers – *fast* and *slow* – taking times t_F and t_S respectively, with $t_F < t_S$. More specifically, assume that motion is the only slow controller. The set of system propositions is partitioned based on the speed of the corresponding low-level controllers, into $\mathcal{Y}_S = \mathcal{L}$ and $\mathcal{Y}_F = \mathcal{Y} \backslash \mathcal{L}$ (i.e. location and non-location propositions). In Example 6, $\mathcal{Y}_F = \{\pi_{camera}\}$ and $\mathcal{Y}_S = \{\pi_{r_1}, \pi_{r_2}\}$.

The approach to continuous execution in 6.1 has two undesirable qualities when actions are non-instantaneous (i.e. violate the assumption made at synthesis time):

Delayed Response

The continuous execution of 6.1 involves completing the "slow actions" (in this case motion) first. When slow actions are non-instantaneous, this execution can result in a perceived lack of responsiveness since it is usually desirable to respond to sensor inputs as soon as they occur. For example, the camera should be turned on as soon as a person is sensed, regardless of the other actions to be performed. However for transition (q_0, q_1) in Figure 6.2, using the approach to continuous execution in 6.1, the robot will not turn on its camera as soon as it senses a person, instead waiting until the transition to r_2 has been completed.

Unsafe Intermediate States

On the other hand, consider what happens when violating the assumption that non-motion (i.e. "fast") controllers have instantaneous execution. While continuous execution in 6.1 is safe for instantaneous fast actions, it admits potentially unsafe intermediate

states when fast actions are non-instantaneous. For example, if the low-level controller for turning on the camera is non-instantaneous, the continuous execution of the controller in Example 6 will pass through the intermediate state $q_0^{\bar{f}s}$ (not present in the discrete automaton) with $\gamma_{\mathcal{Y}}(q_0^{\bar{f}s}) = \{\pi_{r_2}\}$, as depicted in Figure 6.3(b). Although in this example, $q_0^{\bar{f}s}$ does not violate the specification, this may not be true in general, as demonstrated by Example 7 below.

To address the problem of delayed response, it is desirable to be able to activate the camera simultaneously with the motion to allow immediate reaction to the person sensor (unlike the continuous execution paradigm in [26]). This section therefore considers a continuous execution paradigm where all action controllers for a given transition are activated at the same time.

Example 7. Consider Example 6 with the added safety:

$\square(\neg(\pi_{camera} \wedge \pi_{r_1}))$ (Do not activate the camera in r1)

Example 7 requires that the camera never be turned on when in r_1. In this case, the execution resulting from turning on the camera and starting the motion to r_2 at the same time would pass through the intermediate state $q_0^{f\bar{s}}$ (Figure 6.3(c)), not present in the discrete automaton. This execution would therefore be unsafe. Note that the camera turning on in r_1 is not captured by the discrete (and safe) model. Note that, for this example, the execution depicted in Figure 6.3(b) is still safe, while that in Figure 6.3(c) is not. It is desirable to obtain a controller that guarantees safety of intermediate states like $q_0^{f\bar{s}}$, which are not explicitly present in the synthesized automaton or checked during the existing synthesis process, but rather occur as artifacts of the continuous execution.

It may seem reasonable to circumvent these problems by requiring at most one robot action per transition; however, this could result not just in unnecessarily large automata,

but also in newly unsynthesizable specifications. For instance, the specification in Example 6 would be unsynthesizable because the robot can never move from r_1 to r_2 if the environment alternates between person and no person (since the robot will have to toggle the camera on and off, and cannot move while doing so).

Let (q, q') be a potential discrete transition in the non-deterministic automaton representing the problem. Given $\mathcal{Y} = \mathcal{Y}_F \cup \mathcal{Y}_S$, define $\gamma_{\mathcal{Y}_F}(q) = \gamma_{\mathcal{Y}}(q) \cap \mathcal{Y}_F$ and $\gamma_{\mathcal{Y}_S}(q) = \gamma_{\mathcal{Y}}(q) \cap \mathcal{Y}_S$. Let $q^{f_{\bar{s}}(q')}$ denote the discrete state with $\gamma_{\mathcal{Y}}(q^{f_{\bar{s}}(q')}) = \gamma_{\mathcal{Y}_S}(q) \cup \gamma_{\mathcal{Y}_F}(q')$. This is the intermediate state in the transition between q and q', such that the fast actions have finished executing but the slow actions have not. Define

$$h(q, q') = \begin{cases} qq^{f_{\bar{s}}(q')} & \text{if } \gamma_{\mathcal{Y}_S}(q) \neq \gamma_{\mathcal{Y}_S}(q') \wedge \gamma_{\mathcal{Y}_F}(q) \neq \gamma_{\mathcal{Y}_F}(q') \\ q & \text{otherwise} \end{cases}$$

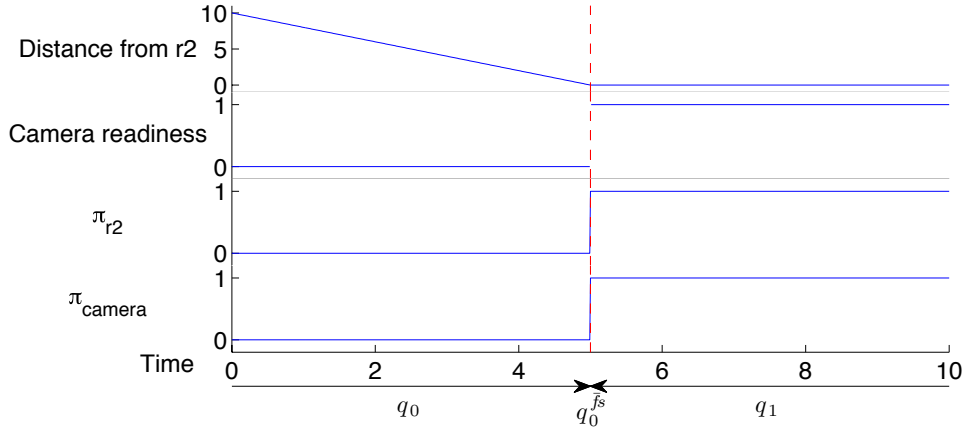
This is a function that returns the intermediate state $q^{f_{\bar{s}}(q')}$ if both slow and fast actions change over the transition (q, q'). Note that if only the slow actions or only the fast actions change, there is no intermediate state in the continuous execution. In Example 6, $h(q_0, q_1) = q_0 q_0^{f_{\bar{s}}(q')}$.

Definition 5. Given $A = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_{\mathcal{X}}, \gamma_{\mathcal{Y}})$, let

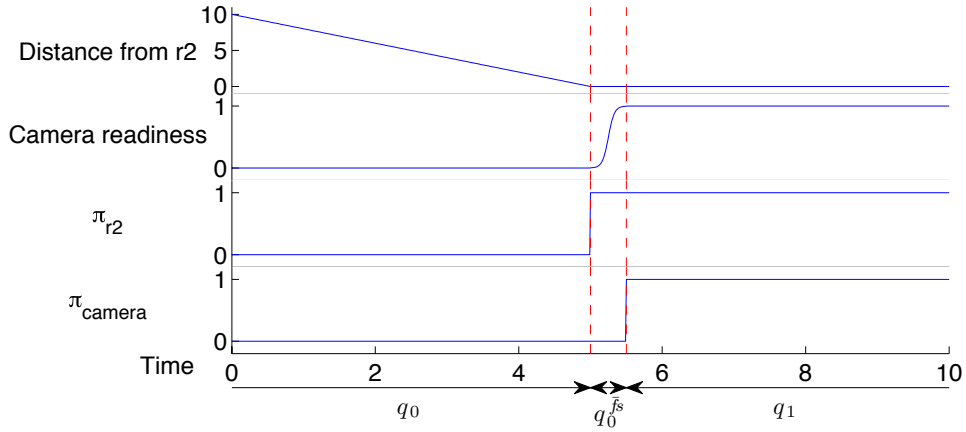
$$H_A^{FS} = \{h(q_0, q_1)...h(q_i, q_{i+1})... \mid q_0 q_1 ... \in Q^\omega, q_{i+1} \in \delta(q_i)\}.$$

H_A^{FS} defines the projection onto the set of discrete states Q of all continuous executions of automaton A when there are controllers of two completion times, and they are executed simultaneously to implement each discrete transition.

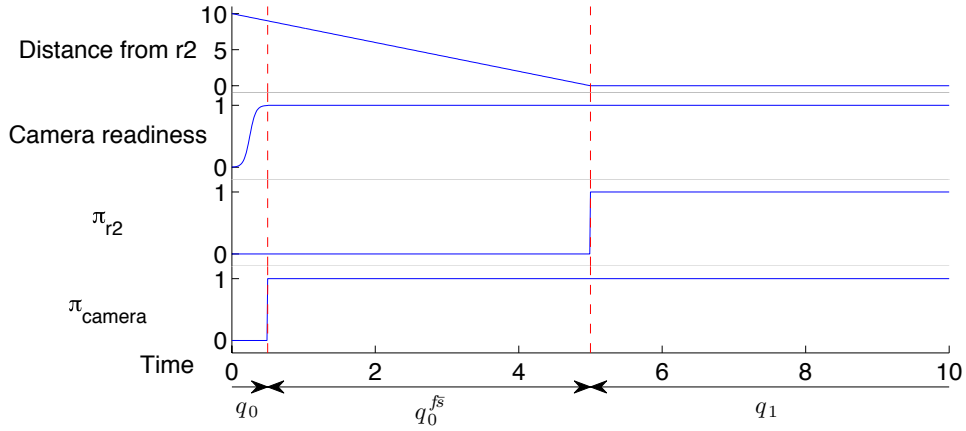
Problem 2. *Given $\varphi, \mathcal{Y} = \mathcal{Y}_F \cup \mathcal{Y}_S$ and a set of safe states Q_{safe}, the goal is to construct A_φ such that $\forall \sigma \in H_{A_\varphi}^{FS}, \sigma \in Q_{safe}^\omega$ (if such an automaton exists).*



(a) Motion completes first, instantaneous camera. This corresponds to the approach in [26] (assuming instantaneous fast actions).



(b) Actual execution corresponding to 6.3(a). Motion completes first, camera is non-instantaneous.



(c) Camera completes first when both controllers are activated together as in [45].

Figure 6.3: Timing diagrams for continuous execution of transition (q_0, q_1) in Figure 6.2

.

Intuitively, the goal is to generate an implementing automaton such that every continuous execution contains only safe states. The set of safe states Q_{safe} can be arbitrarily defined – in this work, it is the set of all states not explicitly excluded by the specified safety properties φ_e^t and φ_s^t.

6.2.2 Solution

In response to the above problem, [45] presents a synthesis algorithm and continuous execution paradigm that guarantees correctness of continuous executions when simultaneously executing low-level controllers of two different completion times for every discrete transition. That work presents a framework that explicitly introduces at the discrete level the intermediate states arising during continuous execution in order to formally reason about them.

The synthesis is based on the algorithm in [37], which reduces the realizability problem to finding a winning strategy in a game played between the system (robot) and the (adversarial) environment. The two players alternate "moves", which correspond to setting values for their respective propositions according to their transition relations: the environment moves first in each time step, and is followed by the system. An infinite alternation of player moves is winning for the system if it either satisfies the system transition relation and liveness requirements, or prevents the environment assumptions from being fulfilled. The set of states from which there exists a winning strategy for the system is called the winning set of states. If the specification is realizable, every initial state admitted by $\varphi_s^i \Rightarrow \varphi_e^i$ is winning for the system. On the other hand, if the environment can make moves to prevent the system from responding in a manner that satisfies φ_s, no automaton is generated.

As mentioned in Chapter 3, finding a winning strategy for the robot in the above game can be thought of as exploring the nondeterministic automaton N_φ in the aim of finding a deterministic automaton "contained" in it, that realizes the specification. The existence of such a discrete automaton can be formally determined using the modal μ-calculus, which extends propositional modal logic with least and greatest fixpoint operators μ, ν [18].

Given a set of propositions P, $P \models \varphi$ denotes that truth assignments setting $\pi \in P$ to `True` and $\pi \notin P$ to `False` satisfy the Boolean formula φ. The semantics of μ-calculus formulae over A_φ is defined recursively:

- A Boolean formula φ is interpreted as the set of states $[\![\varphi]\!]$ in which φ is true, i.e. $[\![\varphi]\!] = \{q \in Q \mid \gamma(q) \models \varphi\}$. The set of states $[\![\varphi]\!]$ is defined inductively on the structure of the μ-calculus formula.
- The logical operator $\circledcirc$ is defined as in [37]:
  $[\![\circledcirc\, \varphi]\!] = \{q \in Q \mid \forall x \in \delta_{\mathcal{X}}(q),\, \delta(q, x) \cap [\![\varphi]\!] \neq \emptyset\}$. In words, this is the set of states q from which the system can force the play to reach a state in $[\![\varphi]\!]$, regardless of what move the environment makes from q (i.e. for every $x \in \delta_{\mathcal{X}}(q)$). In Example 6, $[\![\circledcirc\, \pi_{r_2}]\!]$ is the set of all states in which the robot can move to region r_2, regardless of what the environment does, so $[\![\circledcirc\, \pi_{r_2}]\!] = \{q_0, q_1, q_2\}$ in Figure 6.2.
- Let $\psi(X)$ denote a μ-calculus formula ψ with free variable X. $[\![\mu X.\psi(X)]\!] = \cup_i X_i$ where $X_0 = \emptyset$ and $X_{i+1} = [\![\psi(X_i)]\!]$. This is a least fixpoint operation, computing the smallest set of states X satisfying $X = \psi(X)$.
- $[\![\nu X.\psi(X)]\!] = \cap_i X_i$ where $X_0 = Q$ and $X_{i+1} = [\![\psi(X_i)]\!]$. This is a greatest fixpoint operation, computing the largest set of states X satisfying $X = \psi(X)$.

In [37], the set of winning states for the system is characterized by the μ-calculus formula $\varphi_{win} =$

$$\nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} . \begin{bmatrix} \mu Y.(\bigvee_{i=1}^{m} \nu X.(J_s^1 \wedge \textcircled{\scriptsize\diamond} Z_2 \vee \textcircled{\scriptsize\diamond} Y \vee \neg J_e^i \wedge \textcircled{\scriptsize\diamond} X)) \\ \mu Y.(\bigvee_{i=1}^{m} \nu X.(J_s^2 \wedge \textcircled{\scriptsize\diamond} Z_3 \vee \textcircled{\scriptsize\diamond} Y \vee \neg J_e^i \wedge \textcircled{\scriptsize\diamond} X)) \\ \vdots \\ \mu Y.(\bigvee_{i=1}^{m} \nu X.(J_s^n \wedge \textcircled{\scriptsize\diamond} Z_1 \vee \textcircled{\scriptsize\diamond} Y \vee \neg J_e^i \wedge \textcircled{\scriptsize\diamond} X)) \end{bmatrix}$$

where J_e^i is the i^{th} environment liveness ($i \in \{1, ..., m\}$), and J_s^j is the j^{th} system liveness ($j \in \{1, ..., n\}$). Let $\oplus$ denote summation modulo n. For $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$, the greatest fixpoint $\nu X.(J_s^j \wedge \textcircled{\scriptsize\diamond} Z_{j\oplus 1} \vee \textcircled{\scriptsize\diamond} Y \vee \neg J_e^i \wedge \textcircled{\scriptsize\diamond} X)$ characterizes the set of states from which the robot can force the game to stay infinitely in states satisfying $\neg J_e^i$, thus falsifying the left-hand side of the implication $\varphi_e \Rightarrow \varphi_s$, or in a finite number of steps reach a state in the set $Q_{win} = [\![J_s^j \wedge \textcircled{\scriptsize\diamond} Z_{j\oplus 1} \vee \textcircled{\scriptsize\diamond} Y]\!]$. The two outer fixpoints ensure that the robot wins from the set Q_{win}: μY ensures that the play reaches a $J_s^j \wedge \textcircled{\scriptsize\diamond} Z_{j\oplus 1}$ state in a finite number of steps, and νZ ensures that the robot can loop through the livenesses in cyclic order. From the intermediate steps of the above computation, it is possible to extract an automaton that realizes the specification, provided every initial state is winning; details are available in [37].

To incorporate the relative execution times of the robot controllers, the synthesis algorithm is further constrained to generate only automata with safe intermediate states as follows. Given $\varphi, \mathcal{Y} = \mathcal{Y}_F \cup \mathcal{Y}_S$ and Q_{safe}, define the operator:

$$
\begin{aligned}
[\![\diamondsuit_{FS} \varphi]\!] = \quad \{ \quad & q \in Q \mid \forall x \in \delta_{\mathcal{X}}(q), \\
& \text{either} \\
& \quad \exists q' \in \delta(q, x) \cap [\![\varphi]\!] \text{ such that} \\
& \quad (\gamma_{\mathcal{Y}_F}(q) = \gamma_{\mathcal{Y}_F}(q') \text{ or } \gamma_{\mathcal{Y}_S}(q) = \gamma_{\mathcal{Y}_S}(q')), \\
& \text{or} \\
& \quad \exists q' \in \delta(q, x) \cap [\![\varphi]\!] \text{ such that} \\
& \quad \gamma_{\mathcal{Y}_F}(q) \neq \gamma_{\mathcal{Y}_F}(q') \text{ and } \gamma_{\mathcal{Y}_S}(q) \neq \gamma_{\mathcal{Y}_S}(q'), \\
& \quad \text{and } q^{f\bar{s}} \in Q_{safe} \quad \}
\end{aligned}
$$

This is the set of states from which the system can in a single step force the play to reach a state in $[\![\varphi]\!]$, either by executing actions of only one controller duration (fast or slow), or by executing actions of both fast and slow controller durations. In the former case, there are no intermediate discrete states in the continuous execution; in the latter case, the intermediate state $q^{f\bar{s}}$ is safe.

Informally, $\diamondsuit_{FS}$ is the set of states q from which the system can force the play to reach a state in $[\![\varphi]\!]$, regardless of what move the environment makes from q, with the additional constraint that, if both fast and slow controllers are to be executed to implement a transition, the resulting intermediate state $q^{f\bar{s}}$ in Example 6 (depicted in Figure 6.4) is safe.

6.2.3 Continuous Execution

The proposed synthesis algorithm is accompanied by a new execution paradigm that calls all low-level controllers corresponding to a discrete transition simultaneously. Thus, to execute the transition (q_0, q_1), the hybrid controller constructed for Example 6 activates the controller for turning on the camera (fast) simultaneously with that for

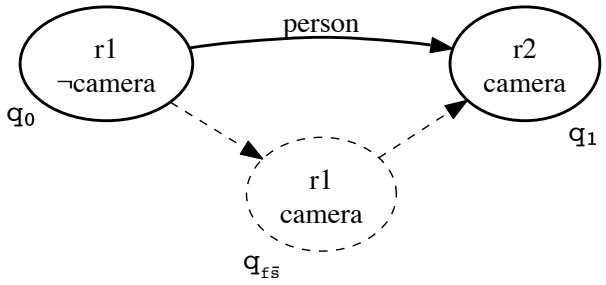


Figure 6.4: Intermediate state with fast camera and slow motion for transition (q_0, q_1) in Figure 6.2

moving from r_1 to r_2 (slow). The transition (q_0, q_1) is completed only when the motion completes.

Returning to Example 7, where the system safety condition includes "Always not camera in r_1" ($\square(\neg(\pi_{camera} \wedge \pi_{r_1}))$), a state in which the system senses a person is only in $[\![\diamondsuit_{FS}\, \pi_{camera}]\!]$ if the robot can stay in the same region while turning on the camera. Recall that q_0 is the state in which the robot is in r_1 with the camera off. Observe that $q_0 \notin [\![\diamondsuit_{FS}\, \pi_{camera}]\!]$ (this means that in q_0, the robot cannot guarantee that the camera will be turned on in the next time step). This is because, if the environment sets π_{person} to true while the robot is still in r_1, the safety condition $\square(\neg(\pi_{camera} \wedge \pi_{r_1}))$ prevents the robot from turning on the camera before first moving to r_2, and so the camera cannot be immediately activated since it might finish execution before the robot had left r_1. The corresponding specification is unrealizable under the new synthesis algorithm, whereas the original synthesis algorithm would return an automaton that included the transition (q_0, q_1) in Figure 6.2. This difference is consistent with the observation that this transition is safe for the original execution in [26], under the assumption of instantaneous fast actions, but is unsafe if all action controllers are to be called simultaneously.

Consider again the specification in Example 6, in which the robot has to move from

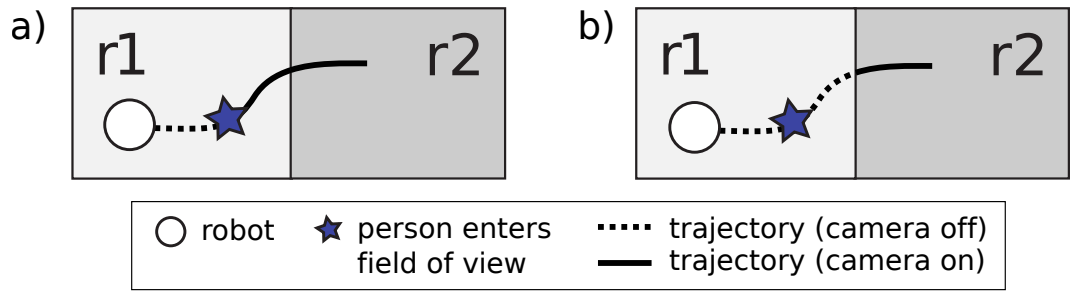


Figure 6.5: Comparison of continuous trajectories and discrete events resulting from the two approaches for Example 6. a) Camera is turned on as soon as a person is sensed, according to the approach in 6.2. b) When a person is sensed, motion is completed first, then camera turns on. This corresponds to the approach in 6.1.

its starting position r_1 to its destination r_2 and turn its camera on if it sees a person along the way. With the new execution paradigm, the hybrid controller turns the camera on immediately when a person is sensed. The trajectory that results from this controller is depicted in Figure 6.5(a). Using the execution paradigm in 6.1, where slow actions are executed before fast actions, this specification would result in undesired behavior: even if the robot sensed a person while in the middle of r_1, it would only react to it once it completed its movement to region r_2. This is depicted in Figure 6.5(b). Furthermore, the person would still need to be sensed at the time of region transition, or else a different transition would be chosen and the person would effectively be ignored.

However, since transitions are now explicitly non-instantaneous, the execution paradigm ignores changes in the environment once a transition has been started (i.e. following activation of a fast controller), until the destination state is reached. This approach therefore produces controllers that are correct under the assumption that the environment does not change during the execution of a discrete transition; any inputs that violate this assumption will be ignored. In the above example, any changes in the environment once the camera has turned on will be ignored until the motion completes.

6.3 Relaxing Assumptions on Relative Action Completion Times

As described in Section 6.2.1, the controllers generated in 6.1 make the assumption of instantaneous actions, and can result in unsafe executions when this assumption is violated. In addition, even with this assumption, the continuous executions exhibit delayed response to the environment. In contrast, the correctness of the controllers generated in 6.2.2 relies on the assumption that the environment does not change during the execution of a discrete transition; violating this assumption also results in unsafe executions as demonstrated by the example below.

Consider again the transition (q_0, q_1) in Figure 6 where the camera will be turned on immediately in response to sensing a person (at the same time as motion), but will complete before the robot has reached r_2. The camera is thus immediately responsive to the person. However, suppose the robot stops sensing a person after the camera has been turned on, but before it has reached r_2. Then the transition (q_0, q_1) will be aborted (i.e. no longer be taken), and the transition (q_0, q_2) will be taken instead. This results in the camera going from on to off, violating the safety condition that enforces persistence of the camera action.

To avoid such unsafe behaviors, the execution paradigm proposed in 6.2.2 will ignore the disappearance of a person after the camera has turned on. Correctness is therefore at the expense of being fully responsive to the environment during the time taken to move between regions. Additionally, the approach in 6.2 assumes a known ordering on action completion times, reducing the number of intermediate states to be checked. Extending the approach to an unknown ordering on action completion times leads to an exponential blow-up, due to the combinatorial number of intermediate states that must be considered and checked for safety.

Relaxing the assumptions on the continuous execution made by the previous two approaches leads to the following problem.

Problem 3. *Given a specification φ and a set of actions with unknown relative completion times, construct an automaton such that every continuous execution satisfies φ while allowing immediate reactivity as well as continual responsiveness to changes in the environment.*

6.4 Provably Correct Controllers for Arbitrary Relative Execution Durations [50]

This section proposes an alternative framework that allows immediate reactivity as well as continual responsiveness to changes in the environment, and generalizes directly to arbitrary (but finite) action completion times. The continuous execution also relaxes the previous assumptions on the low level controller execution durations, with a small computational overhead. To account for the non-instantaneous execution of continuous controllers, each robot action is viewed as the *activation* of the corresponding low level controller, and a new sensor proposition is introduced in the discrete model to indicate whether the controller has completed execution. That is, the robot is able to sense when a low level controller has completed its action (e.g., the camera has turned on, or it has arrived in region r_1).

6.4.1 Discrete Abstraction

The set of propositions is now modified to consist of:

- π_s for each sensor input s (e.g., π_{person} is true if and only if a person is sensed)
- π_a for the *activation* of each robot action a (e.g., π_{camera} is true if and only if the robot has activated the controller to turn on the camera). Similarly, $\neg\pi_a$ represents the activation of the controller for turning a off.
- π_r for the *initiation* of motion towards each region r (e.g., $\pi_{bedroom}$ is true if and only if the robot is trying to move to the bedroom). φ_r is defined as in Chapter 3.
- π_a^c, π_r^c for the completion of the controller for turning action a on, or motion to region r (e.g., $\pi_{bedroom}^c$ is true if and only if the robot has arrived in the bedroom, and π_{camera}^c is true if and only if the camera has finished turning on). $\neg\pi_a^c$ represents the completion of the controller for turning action a off.[1]

Action/motion completion is modeled as an event sensed by the robot, and therefore $\mathcal{X} = \pi_a^c \cup \pi_r^c \cup \pi_s, \mathcal{Y} = \pi_a \cup \pi_r$. For Example 6, $\mathcal{X} = \{\pi_{person}, \pi_{r_1}^c, \pi_{r_2}^c, \pi_{camera}^c\}$ and $\mathcal{Y} = \{\pi_{r_1}, \pi_{r_2}, \pi_{camera}\}$.

6.4.2 Formal Specification Transformation

Given this discrete abstraction, the task specification must be rewritten to govern both which actions can be activated by the robot, and how the action-completion sensors behave.

Proposition Replacement in Original Specification

Task specification $\varphi = (\varphi_e \Rightarrow \varphi_s)$ in the framework of [26] is modified as follows:

[1]Note that this work considers actions other than motion to have on and off modes only, but the approach extends to other types of actions. For example, the intermediate stages of the camera turning on could be modeled separately, such as sensor cleaning, battery check, detecting external memory, etc.

English specification	**Original LTL (φ)**	**New LTL (φ')**
Robot starts in region r_1 with the camera off	$\varphi_{r_1} \wedge \neg\pi_{camera}$	$\pi^c_{r_1} \wedge \neg\pi^c_{camera}$
Activate the camera if you see a person	$\square(\bigcirc \pi_{person} \Rightarrow \bigcirc \pi_{camera})$	$\square(\bigcirc \pi_{person} \Rightarrow \bigcirc \pi_{camera})$
Camera stays on once turned on	$\square((\pi_{camera} \Rightarrow \bigcirc \pi_{camera}))$	$\square((\pi^c_{camera} \Rightarrow \bigcirc \pi_{camera}))$
Go to r2 infinitely often	$\square\diamondsuit(\pi_{r_2})$	$\square\diamondsuit(\pi^c_{r_2})$

Table 6.1: Replacing propositions in the task specification for Example 6

- Initial conditions specify the sensed state of the robot, so every occurrence of π_a and π_r in φ^i_s is replaced with π^c_a and π^c_r respectively.
- Robot goals are predicated on the completion of actions (as sensed by the corresponding sensor). So every occurrence of π_a in φ^g_s is replaced with π^c_a. Similarly, robot goals refer to the sensed location π^c_r rather than just the activation of the motion controller π_r.
- Robot safety conditions govern which controllers are to be activated in response to events in the environment, and may refer to the sensing of action completion as well as events in the environment. The user input language, such as that presented in [25], must therefore allow distinguishing between a reference to π_a and π^c_a in safety conditions, as discussed in 6.5.2.

The resulting LTL specification is denoted $\hat{\varphi} = \hat{\varphi}_e \Rightarrow \hat{\varphi}_s$. Table 6.1 presents the LTL formulas corresponding to the specification for Example 6, provided in Section 6.1, before and after proposition replacement.

Robot Transition Relation

The allowed robot motion now depends on the sensed location. Given a region r, let $Adj(r)$ denote the set of regions adjacent to r (including r itself). φ_{trans} in 6.1 then

changes as follows:

$$\hat{\varphi}_{trans} = \bigwedge_r \Box(\pi_r^c \Rightarrow \bigvee_{r' \in Adj(r)} \bigcirc \varphi_{r'})$$

For Example 6,

$$\hat{\varphi}_{trans} = \Box(\pi_{r_1}^c \Rightarrow (\bigcirc \varphi_{r_1} \vee \bigcirc \varphi_{r_2}))$$
$$\wedge \Box(\pi_{r_2}^c \Rightarrow (\bigcirc \varphi_{r_2} \vee \bigcirc \varphi_{r_1}))$$

Here φ_{r_1} indicates that the robot is activating the controller to move towards r_1 and not activating the controller to move towards r_2 (i.e. $\varphi_{r_1} = \pi_{r_1} \wedge \neg\pi_{r_2}$). φ_{r_1} is defined symmetrically. The first conjunct in the above transition formula specifies that when the robot is in r_1 or r_2 (i.e. $\pi_{r_1}^c$ or $\pi_{r_2}^c$ is true, respectively), it can either activate the controller for moving towards r_1 or that for moving towards r_2 (since the two regions are adjacent).

Sensor Assumptions

In addition, sensor assumptions are required to define the effects of the robot activating its various controllers:

$$\varphi_c = \bigwedge_r \Box(\pi_r^c \Leftrightarrow \bigwedge_{r' \neq r} \neg\pi_{r'}^c) \tag{6.1}$$
$$\wedge \bigwedge_r \bigwedge_{r' \in Adj(r)} \Box(\pi_r^c \wedge \varphi_{r'} \Rightarrow (\bigcirc \pi_r^c \vee \bigcirc \pi_{r'}^c)) \tag{6.2}$$
$$\wedge \bigwedge_a \Box(\pi_a^c \wedge \pi_a \Rightarrow \bigcirc \pi_a^c) \tag{6.3}$$
$$\wedge \bigwedge_a \Box(\neg\pi_a^c \wedge \neg\pi_a \Rightarrow \bigcirc \neg\pi_a^c) \tag{6.4}$$

Conjunct (6.1) enforces mutual exclusion between the physical locations of the robot. Conjunct (6.2) governs how the location of the robot can change in a single time step in response to the activation of the motion controllers. Conjuncts (6.3-4) govern the completion of other actions in response to the activation of the corresponding controllers. In

the case of Example 6,

$$\begin{aligned}
\varphi_c \;=\; & \Box(\pi^c_{r_1} \Leftrightarrow \neg\pi^c_{r_2}) \wedge \Box(\pi^c_{r_2} \Leftrightarrow \neg\pi^c_{r_1}) && (6.5)\\
& \wedge \Box(\pi^c_{r_1} \wedge \varphi_{r_1} \Rightarrow \bigcirc \pi^c_{r_1}) && (6.6)\\
& \wedge \Box(\pi^c_{r_1} \wedge \varphi_{r_2} \Rightarrow \bigcirc \pi^c_{r_1} \vee \bigcirc \pi^c_{r_2}) && (6.7)\\
& \wedge \Box(\pi^c_{r_2} \wedge \varphi_{r_2} \Rightarrow \bigcirc \pi^c_{r_2}) && (6.8)\\
& \wedge \Box(\pi^c_{r_2} \wedge \varphi_{r_1} \Rightarrow \bigcirc \pi^c_{r_2} \vee \bigcirc \pi^c_{r_1}) && (6.9)\\
& \wedge \Box(\pi^c_{camera} \wedge \pi_{camera} \Rightarrow \bigcirc \pi^c_{camera}) && (6.10)\\
& \wedge \Box(\neg\pi^c_{camera} \wedge \neg\pi_{camera} \Rightarrow \neg \bigcirc \pi^c_{camera}) && (6.11)
\end{aligned}$$

For example, conjunct (6.7) states that if the robot is in r_1 (i.e. $\pi^c_{r_1}$ is true) and is activating the controller to move to r_2 (i.e. φ_{r_2}), then in the next time step, the robot is either still in r_1 ($\pi^c_{r_1}$ is true) or has reached r_2 ($\pi^c_{r_2}$ is true). Conjunct (6.10) states that if the camera is already on and is supposed to be on, it will stay on.

Fairness Conditions

In addition to the above safety conditions, additional constraints on the environment are required to ensure that every action/motion eventually completes, i.e. that the robot's environment is in some sense "fair". A first approach is to add an environment assumption that every controller activation or deactivation eventually results in completion, i.e. the fairness conditions

$$\Box \Diamond (\pi_a \Rightarrow \bigcirc \pi^c_a) \tag{6.12}$$

and

$$\Box \Diamond (\neg\pi_a \Rightarrow \bigcirc \neg\pi^c_a) \tag{6.13}$$

for every action a or region r.

However, this adds two fairness assumptions to the specification for every action. Since the synthesis algorithm scales polynomially with the number of fairness assumptions [37], it is important to minimize the number of added assumptions. This can be achieved by introducing a single fairness condition that incorporates the possibility that the robot is forced to "change its mind" by events in the environment. For this purpose, two new Boolean formulas $\varphi_a^{completion}$ and φ_a^{change} are defined for each action a as follows:

$$\varphi_a^{completion} = (\pi_a \wedge \bigcirc \pi_a^c) \vee (\neg\pi_a \wedge \neg \bigcirc \pi_a^c)$$
$$\varphi_a^{change} = (\pi_a \wedge \neg \bigcirc \pi_a) \vee (\neg\pi_a \wedge \bigcirc \pi_a)$$

Formula $\varphi_a^{completion}$ holds when the activation (or de-activation) of the controller for a has completed execution. Formula φ_a^{change} captures the robot changing its mind (such as by toggling the camera). A single pair of formulas $\varphi_{loc}^{completion}, \varphi_{loc}^{change}$ suffices for motion since locations are mutually exclusive and the robot cannot try to move to two locations at once:

$$\varphi_{loc}^{completion} = \bigvee_r (\varphi_r \wedge \bigcirc \pi_r^c))$$
$$\varphi_{loc}^{change} = \bigvee_r (\varphi_r \wedge \neg \bigcirc \varphi_r))$$

The complete fairness assumption added is:

$$\varphi_{fair}^a = \Box \diamondsuit (\varphi_a^{completion} \vee \varphi_a^{change}) \tag{6.14}$$

Note that every execution satisfying both fairness conditions (6.12) and (6.13) described above for activation and deactivation also satisfies (6.14). Moreover, there is only one such assumption added for each action a (in the case of Example 6, there is one such assumption for the camera). Additionally, there is one assumption φ_{fair}^{loc} for motion. For Example 6,

$$\begin{aligned}
\varphi_{fair}^{camera} &= \Box \diamondsuit [(\pi_{camera} \wedge \bigcirc \pi_{camera}^c) \vee (\neg\pi_{camera} \wedge \neg \bigcirc \pi_{camera}^c) \\
&\quad \vee (\pi_{camera} \wedge \neg \bigcirc \pi_{camera}) \vee (\neg\pi_{camera} \wedge \bigcirc \pi_{camera})] \\
\varphi_{fair}^{loc} &= \Box \diamondsuit [(\pi_{r_1} \wedge \bigcirc \pi_{r_1}^c) \vee (\pi_{r_2} \wedge \bigcirc \pi_{r_2}^c) \\
&\quad \vee (\pi_{r_1} \wedge \neg \bigcirc \pi_{r_1}) \vee (\pi_{r_2} \wedge \neg \bigcirc \pi_{r_2})]
\end{aligned}$$

Given a task specification $\varphi = (\varphi_e \Rightarrow \varphi_s)$, the LTL specification used to synthesize a controller (after proposition replacement, changing the robot transition relation, and adding sensor assumptions and fairness conditions) is now :

$$\varphi_{new} = \hat{\varphi}_e \wedge \varphi_c \wedge \bigwedge_a \varphi^a_{fair} \wedge \varphi^{loc}_{fair} \Rightarrow \hat{\varphi}_s \wedge \hat{\varphi}_{trans}$$

6.4.3 Synthesis

Since the formulas $\varphi_a^{completion}$ and φ_a^{change} in the proposed liveness condition φ^a_{fair} govern both current and next time steps, the original synthesis algorithm in [37] cannot be applied as-is to synthesize an implementing automaton for the specification. Liveness conditions that incorporate temporal formulas with both current and next time steps are handled by changing the computation of the set of robot-winning states as follows.

Define $\diamondsuit$ such that $[\![\diamondsuit\varphi]\!] = \{q \in Q \mid \forall x \in \delta_{\mathcal{X}}(q), \delta(q,x) \in [\![\bigcirc \varphi]\!]\}$ if φ contains no primed environment or robot variables, otherwise $[\![\diamondsuit\varphi]\!] = [\![\diamondsuit \varphi]\!]$. Then the new set of winning states is defined as:

$$\hat{\varphi_{win}} = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} . \begin{bmatrix} \mu Y. (\bigvee_{i=1}^m \nu X. \diamondsuit'(J_s^1 \wedge Z_2' \vee Y' \vee \neg J_e^i \wedge X')) \\ \mu Y. (\bigvee_{i=1}^m \nu X. \diamondsuit'(J_s^2 \wedge Z_3' \vee Y' \vee \neg J_e^i \wedge X')) \\ \vdots \\ \mu Y. (\bigvee_{i=1}^m \nu X. \diamondsuit'(J_s^n \wedge Z_1' \vee Y' \vee \neg J_e^i \wedge X')) \end{bmatrix}$$

The only difference from φ_{win} in Section 6.2.2 is to replace $(J_s^i \wedge \diamondsuit Z_{(i+1) \bmod n} \vee \diamondsuit Y \vee \neg J_e^i \wedge \diamondsuit X)$ with $\diamondsuit'(J_s^i \wedge Z'_{(i+1) \bmod n} \vee Y' \vee \neg J_e^i \wedge X')$. Note that if none of the robot liveness conditions contain the $\bigcirc$ operator, the computations of φ_{win} and $\hat{\varphi_{win}}$ are identical.

The intermediate stages of this computation now compute states that can force goal

transitions (rather than reach goal *states*). The strategy extraction has to accordingly be changed to accommodate goal transitions rather than goal states. As in [37], there are three types of transitions. Each transition of type ρ_1 is a J_s^j winning transitions for some j, and results in the pursued goal changing from j to $j+1$. Transitions of type ρ_2 are taken in the case that we can get closer to a J_s^j transition. Transitions of type ρ_3 falsify some fairness assumption J_e^i, and repeating such a transition forever is a legitimate computation because it violates the environment requirement of infinitely many transitions satisfying J_e^i.

Note that it is possible to use the original synthesis algorithm (which only allows simple Boolean formulas in liveness conditions) to synthesize a controller by introducing a new proposition, $\pi_a^{completion,change}$, and the safety condition

$$\square(\bigcirc \pi_a^{completion,change} \iff (\varphi_a^{completion} \vee \varphi_a^{change}))$$

This allows the additional liveness to instead be written as

$$\varphi'_{fair} = \square \diamondsuit \pi_a^{completion,change}$$

However, this introduces one new proposition per robot action. The complexity of the synthesis algorithm scales with the size of the state space, which in turn scales exponentially with the number of propositions; in addition, extra propositions result in larger automata.

Even with the above change to the synthesis algorithm, one environment proposition must be added per robot action (corresponding to the sensor for action completion). In the worst case, the time taken for synthesis is therefore still increased by a factor of $2^{|\mathcal{Y}|}$ over the original approach. With the original synthesis algorithm, the increase is by a factor of $4^{|\mathcal{Y}|}$ (since two new propositions are required per action).

6.4.4 Continuous Execution

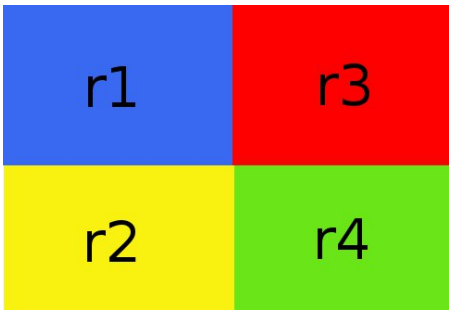
Given a state q, observed sensor values $X \subset \mathcal{X}$ and the corresponding next state q' in the automaton, the transition (q, q') is executed by simultaneously invoking the controllers corresponding to every action or location proposition π_a that changes value from q to q'. Note that the current sensed state of the system, as represented by X, determines which actions a can be activated in q'. Transitions in the automaton are instantaneous, as they correspond to activation or deactivation of controllers, but the controllers themselves may take several discrete transitions to complete execution.

The resulting controller exhibits the desired properties:

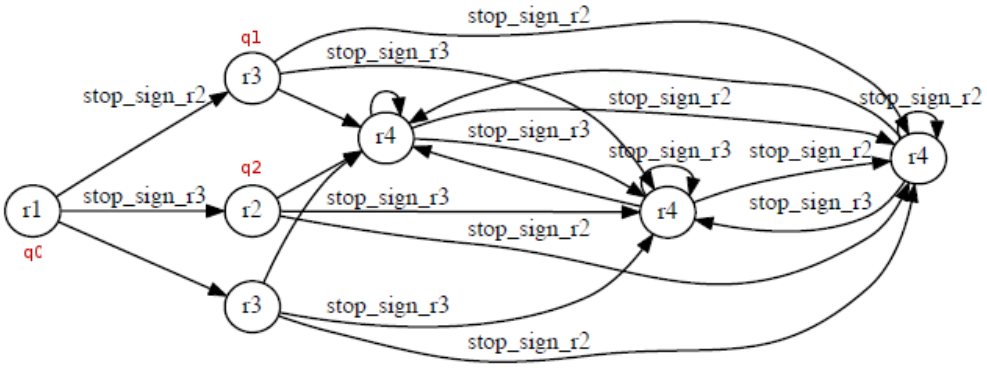
- actions are executed immediately in response to sensor events, eliminating the delayed reactivity of [26];
- safety of continuous executions is guaranteed even when the robot is forced to changes its mind due to changes in the environment;
- the approach extends to any number of robot actions, with arbitrary relative timings, although the computational burden increases for a large number of actions.

6.5 Examples

This section provides examples illustrating the effectiveness of the proposed solution. The robot controllers for the examples presented were synthesized using LTLMoP[11].



(a) Workspace for Example 8



(b) Synthesized automaton using approach in [26]

Figure 6.6: Workspace and original automaton for Example 8. Negated sensor labels are omitted from the transitions for clarity.

6.5.1 Safety of Physical Execution

Figure 6.7 depicts an excerpt of the automaton synthesized for Example 6. The full automaton has 11 states and is omitted for conciseness. Negated sensor labels are omitted from the transitions for clarity. The label `c_r_i` represents $\pi^c_{r_i}$.

In state q_0, the robot is trying to stay in r_1 (as indicated by the action π_{r_1} being true), and not activating its camera (π_{camera} is false); note that $q_0 \in Q_0$. Consider the transition (q_0, q_1), which is activated when the robot is in q_0 and does not sense a person. The robot is still in r_1 (indicated by $\pi^c_{r_1}$ being true on the transition into q_1. It is now trying to go to r_2, indicated by π_{r_2} being true in q_1. When in q_1, if the robot still does

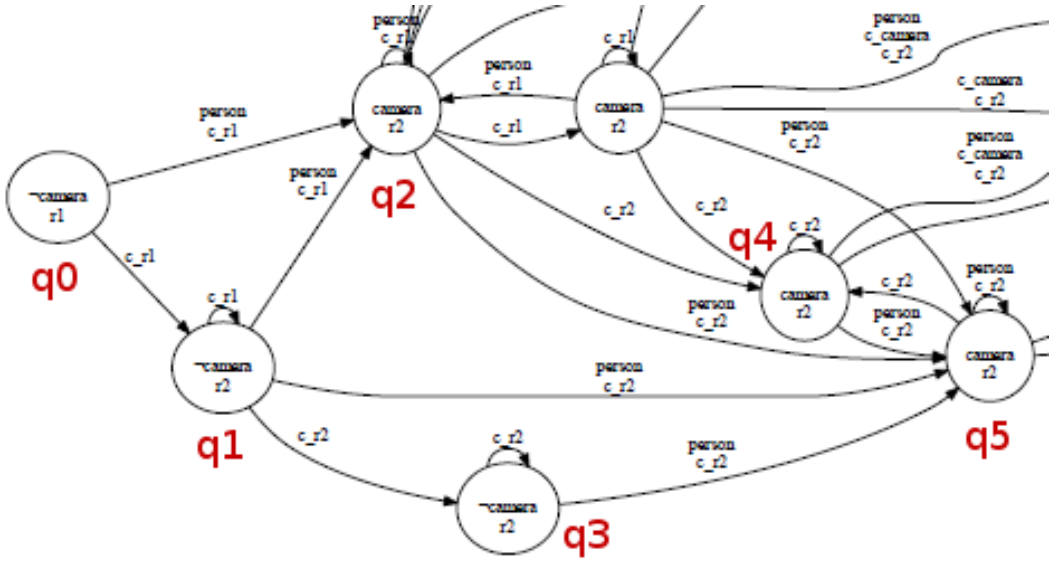


Figure 6.7: Excerpt of automaton synthesized for Example 6 with the approach in 6.4. Negated propositions are omitted from the transitions for clarity.

not sense a person, it either moves to q_3 or stays in q_1 depending on whether it has reached r_2 yet (i.e. depending on the truth value of $\pi^c_{r_2}$). On the other hand, suppose the robot senses a person in q_1 before it has reached r_2, the transition enabled is instead (q_1, q_2). In state q_2, the robot is still activating the motion controller to reach r_2, but is now additionally activating the camera; the transition to q_4 is taken once the robot is in r_2. Finally, the transition (q_1, q_5) is taken if the robot senses a person exactly as it reaches r_2 (as indicated by both sensor propositions π_{person} and $\pi^c_{r_2}$ being true on that transition).

Note that in the continuous execution of the above automaton, all the controllers are invoked at the same time. For example, in the transition (q_0, q_2), which is activated when the robot is in q_0 and senses a person, the controllers for moving from r_1 to r_2 and for turning on the camera are being activated simultaneously. Any difference in their completion times is captured by the corresponding sensor propositions. Even if the person disappears before the motion from r_1 to r_2 is completed, the transition (q_0, q_2) is still taken (followed by a transition out of q_2 that corresponds to the person no longer

being seen), and the camera is still being activated in q_2. This ensures that the person is not ignored, since there is an explicit state representing the fact that the camera is being turned on even though the person has disappeared.

6.5.2 Activation and Completion Dependent Safety Properties

Desired safety properties can now also be defined in terms of which actions have completed rather than in terms of which actions are activated. For example, in order to require that the camera never be turned on in r_1, the system safety can be augmented with either

$$\square(\pi^c_{r_1} \Rightarrow \neg\pi^c_{camera}) \quad \text{or} \quad \square(\pi^c_{r_1} \Rightarrow \neg \bigcirc \pi_{camera}).$$

The first of these requires that the camera not physically be on while the robot is still in r_1, whereas the second specifies that the controller for turning on the camera not be activated while the robot is in r_1. The first option ($\square(\pi^c_{r_1} \Rightarrow \neg\pi^c_{camera})$) could be either a specification of desired behavior or an assumption about the camera controller. Suppose it is a specification. Note that if the robot activates the controller for turning on the camera while it is in r_1, the camera could potentially turn on while the robot is still in r_1. Therefore, both specifications will produce the same observed behavior, where the robot will never turn on the camera in r_1. On the other hand, if it is an included assumption, then the robot is free to activate the controller for the camera in r_1, but according to the assumption, the camera will not turn on.

Recall from Section 6.3 the challenge of ensuring safe execution even when transitions are aborted due to changes in the environment. The safety condition enforcing that the camera stays on once turned on (in Example 6) can be expressed in terms of

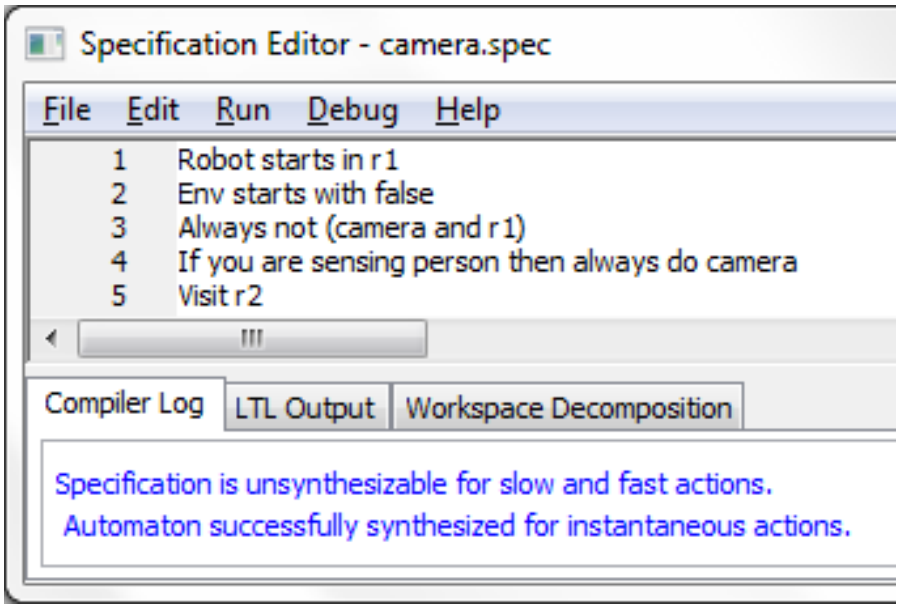


Figure 6.8: Providing feedback on a specification that is unsynthesizable because of the actuation durations [45].

controller activation and completion, as either

$$\square(\pi_{camera} \Rightarrow \bigcirc \pi_{camera}) \quad \text{or} \quad \square(\pi^c_{camera} \Rightarrow \bigcirc \pi_{camera}).$$

The first of these prevents the robot from toggling π_{camera} (such as by flicking the camera switch on and off). The second prevents the robot from trying to turn the camera off once it has sensed that it has turned on; toggling π_{camera} is allowed until the camera has actually turned on. In this manner, the specification can be fine tuned to distinguish between continuous controllers that can be aborted and those that cannot.

6.5.3 Unrealizability Due to Physical Execution

Example 8. Consider the workspace depicted in Figure 6.6(a). The robot starts in r_1 and has to visit r_4. However, if it sees a stop sign in either r_2 or r_3, it cannot pass through that room. A safety assumption on the environment guarantees that there will

never be stop signs in both r_2 and r_3 at the same time. There are initially no stop signs. Given this specification, one might expect an implementing controller that drives the robot from r_1 to r_4 via whichever room (r_2 or r_3) does not have a stop sign. However, if a stop sign appears while the robot is driving to this room, the robot has to turn around and try the other room. If this happens every time the robot starts moving towards r_4, it will never be able to reach r_4. With the approach in [26], the specification for Example 8 is:

$$\wedge \quad \neg\pi_{stop\_sign\_in\_r_2} \wedge \neg\pi_{stop\_sign\_in\_r_3} \qquad \text{\#Env Initial}$$

(Env starts with no stop sign in either r2 or r3)

$$\wedge \quad \square(\neg(\pi_{stop\_sign\_in\_r_3} \wedge \pi_{stop\_sign\_in\_r_2})) \qquad \text{\#Env Safety}$$

(There will never be stop signs in both r2 and r3)

$$\Rightarrow$$

$$\varphi_{r_1} \qquad \text{\#Robot Initial}$$

(Robot starts in r1)

$$\wedge \quad \square(\bigcirc \pi_{stop\_sign\_in\_r_2} \Rightarrow \bigcirc \neg\varphi_{r_2}) \qquad \text{\#Robot Safety}$$

(Do not go to r2 if you sense a stop sign in r2)

$$\wedge \quad \square(\bigcirc \pi_{stop\_sign\_in\_r_3} \Rightarrow \bigcirc \neg\varphi_{r_3}) \qquad \text{\#Robot Safety}$$

(Do not go to r3 if you sense a stop sign in r3)

$$\wedge \quad \square\diamondsuit(\pi_{r_4}) \qquad \text{\#Robot Liveness}$$

(Visit r4 infinitely often)

This specification is realizable under the assumption of instantaneous robot actions, via the synthesis approach in [26], and the synthesized automaton is depicted in Figure 6.6(b). However, consider what happens under the continuous execution paradigm in [26] when the robot is in state q_0 (where it is in r_1), and sees a stop sign in r_2. The robot will start to move towards r_3 (and state q_1). Suppose that before the robot has entered

r_3, the stop sign in r_2 disappears but one appears in r_3. The robot will abort the discrete transition (q_0, q_1) and start heading to r_2 to take the transition (q_0, q_2) instead; note that $\pi_{stop\_sign\_in\_r_2}$ resets over the new transition. If the stop sign's location changes faster than the robot can move, the robot will be trapped in r_1, because it will keep changing its mind between the above two discrete transitions. This is therefore an example of a high-level task that produces a controller under the synthesis approach of [26], but whose physical execution does not accomplish the specified behavior because of an inadequate modeling of the underlying physical system.

With the new discrete abstraction, task specification transformation and execution paradigm presented in this chapter, the robot initial condition in the above specification changes to $\pi^c_{r_1}$, and the robot goal becomes $\square \diamondsuit (\pi^c_{r_4})$. This specification (with the additional formulas introduced in Section 6.4) is unrealizable, and no automaton is obtained. As noted above, this is the safer, more desirable outcome, since there exists an environment strategy that toggles the stop signs between r_2 and r_3 and prevents the robot from fulfilling the specification.

6.6 Explaining Unsynthesizable Specifications

Recent work has addressed the question of providing the user with feedback on a specification that has no implementing controller [38]. It may be the case that a specification is synthesizable in one synthesis framework but unsynthesizable in another. In this situation, the user can be alerted to the fact that the timing semantics of controller execution are responsible for the unsynthesizability of the specification, since unsafe intermediate states may occur. Figure 6.8 shows this feedback being presented to the user in LTLMoP; the specification depicted is unsynthesizable in the framework proposed in 6.2

(i.e. assuming slow and fast actions) because the robot cannot stay in region r_1 while turning on the camera controller if it senses a person. It is, however, synthesizable under the assumption of instantaneous actions using the approach in [26].

Future research will analyze cases of unsynthesizability arising from incorporating timing semantics during controller synthesis, and present users with this information in a useful manner. An additional direction to investigate is the automatic addition of environment assumptions to make the specification synthesizable. For example, in Example 8, adding the environment liveness $\square \diamondsuit (\pi^c_{r_4})$ results in a controller, by explicitly requiring the environment to eventually let the robot through to r_4.

6.7 Conclusions

This chapter describes a challenge of applying formal methods in the physical domain of high-level robot control, namely that of achieving correct continuous behavior from high level specifications when the low-level controllers have different completion times. Three different approaches to timing semantics for controller synthesis are compared, based on the assumptions they make about the execution of low level action controllers. Assumptions range in strength from instantaneous actions, to the case where robot actions are either *fast* or *slow*, to controllers whose relative completion times are unknown. The approaches are compared on factors including the complexity of the resulting synthesis, reactivity to sensor inputs, and the safety of intermediate states arising during execution. Future work includes analyzing specifications that have no implementation because of the timing semantics of the desired controllers, and presenting this information to users.

CHAPTER 7

CONCLUSION

As robot sensing and actuation become more robust, and multi-purpose robots more common, the challenge of achieving provably correct high-level robot control is increasingly important. This dissertation presented solutions to several challenges in ensuring that a user-defined specification yields a robot controller that implements to specified high-level autonomous behavior. The goal of the underlying research is to facilitate the creation of controllers that achieve the behavior intended by the specification-designer.

Chapter 4 provided an algorithm for identifying and explaining the cause of failure in specifications for which there either does not exist an implementing controller, or the implementation is trivial. The algorithm systematically analyzes robot behavior specifications, exploiting the structure of the specification to narrow down possible reasons for failure to create a robot controller. Using this algorithm, the synthesis process is enclosed in a layer of reasoning that identifies the cause of failure, enabling the user to target their attention to the relevant portions of the specification. In addition, the user can explore the cause of unsynthesizability by means of an interactive game.

Chapter 5 builds on the analysis provided by the algorithm in Chapter 4, aiming to provide a minimal explanation for why the robot specification is inconsistent, or how the environment can prevent the robot from fulfilling the desired guarantees. The causes of failure presented are unsynthesizable core subsets of the original specification. A suite of SAT-based techniques is presented for identifying unsatisfiable and unrealizable cores in the case of deadlock and most cases of livelock; iterated realizability checking is used to identify cores in cases where the SAT-based analysis fails. Examples show that the additional analysis provides improvements in terms of reducing the number of sentences in the original specification highlighted, and ignoring irrelevant subformulas.

Future work on analyzing unsynthesizable specifications includes exploiting the environment counterstrategy and other existing analysis techniques to provide the user with more comprehensive forms of feedback, including specific modifications to the specification that would allow synthesis. This may include adding additional environmental assumptions [12, 52, 23] to exclude the specific environments that can prevent the specified robot behavior. Future work also includes automatically determining the depth for obtaining a meaningful core in the case of livelock for the SAT-based approaches, and exploring SAT-based techniques that do not require explicit state extraction of the counterstrategy automaton. Another direction for future study is the empirical comparison of SAT-based techniques with approaches based on iterated realizability testing, to evaluate relative computation time for practical examples.

Finally, Chapter 6 addresses an application-specific challenge of using formal methods in the physical domain of high-level robot control, namely that of achieving correct continuous behavior from high level specifications when the low-level controllers have different completion times. Three different approaches to timing semantics for controller synthesis are compared based on the assumptions they make about the execution of low level action controllers. Assumptions range in strength from instantaneous actions, to the case where robot actions are either *fast* or *slow*, to controllers whose relative completion times are unknown. The approaches are compared on factors including the complexity of the resulting synthesis, reactivity to sensor inputs, and the safety of intermediate states arising during execution. Future work includes analyzing specifications that have no implementation because of the timing semantics of the desired controllers, and presenting this information to users. Additional questions not addressed in this dissertation are the applicability of the presented timing semantics for continuous controller execution to high-level tasks with multiple robots.

BIBLIOGRAPHY

[1] Alessandro Cimatti and Marco Roveri and Viktor Schuppan and Andrei Tchaltsev. Diagnostic Information for Realizability. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 52–67, 2008.

[2] Alessandro Cimatti and Marco Roveri and Viktor Schuppan and Stefano Tonetta. Boolean Abstraction for Temporal Logic Satisfiability. In *Computer Aided Verification (CAV)*, pages 532–546, 2007.

[3] Alur, R. and Henzinger, T.A. and Lafferriere, G. and Pappas, G.J. Discrete Abstractions of Hybrid Systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[4] Amir Pnueli. The Temporal Logic of Programs. In *Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[5] Amir Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 179–190, New York, NY, USA, 1989. ACM.

[6] Amir Pnueli and Yaniv Sa'ar and Lenore D. Zuck. JTLV: A Framework for Developing Verification Algorithms. In *Computer Aided Verification (CAV)*, pages 171–174, 2010.

[7] Amit Bhatia and Lydia E. Kavraki and Moshe. Y. Vardi. Sampling-Based Motion Planning with Temporal Goals. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2689–2696, 2010.

[8] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability (JSAT)*, 4(2-4):75–97, 2008.

[9] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.

[10] Calin Belta and Volkan Isler and George J. Pappas. Discrete Abstractions for Robot Motion Planning and Control in Polygonal Environments. *IEEE Transactions on Robotics*, 21(5):864–874, 2005.

[11] Cameron Finucane and Gangyuan Jing and Hadas Kress-Gazit. LTLMoP: Experimenting with Language, Temporal Logic and Robot Control. In *IEEE/RSJ*

International Conference on Intelligent Robots and Systems (IROS), pages 1988 – 1993, 2010.

[12] Chatterjee, Krishnendu and Henzinger, Thomas A. and Jobstmann, Barbara. Environment Assumptions for Synthesis. In *International Conference on Concurrency Theory (CONCUR)*, pages 147–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Cimatti, Alessandro and Griggio, Alberto and Sebastiani, Roberto. A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 334–339, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[15] Daniel Brooks and Constantine Lignos and Cameron Finucane and Mikhail Medvedev and Ian Perera and Vasumathi Raman and Hadas Kress-Gazit and Mitch Marcus and Holly Yanco. Make It So: Continuous, Flexible Natural Language Interaction with an Autonomous Robot. AAAI Workshops, 2012.

[16] Daniel J. Brooks and Constantine Lignos and Mikhail S. Medvedev and Ian Perera and Cameron Finucane and Vasumathi Raman and Abraham Shultz and Sean McSheehy and Adam Norton and Hadas Kress-Gazit and Mitchell P. Marcus and Holly A. Yanco. Situation Understanding Bot Through Language and Environment. In *Human-Robot Interaction*, pages 419–420, 2012.

[17] David C. Conner and Alfred Rizzi and Howie Choset. Composition of Local Potential Functions for Global Robot Control and Navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 4, pages 3546–3551. IEEE, October 2003.

[18] Dexter Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[20] Eric M. Wolff and Ufuk Topcu and Richard M. Murray. Optimal Control with Weighted Average Costs and Temporal Logic Specifications. In *Robotics: Science and Systems (RSS)*, 2012.

[21] Gangyuan Jing and Cameron Finucane and Vasumathi Raman and Hadas Kress-

Gazit. Correct High-level Robot Control from Structured English. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3543–3544, 2012.

[22] Gangyuan Jing and Hadas Kress-Gazit. Improving the Continuous Execution of Reactive LTL-Based Controllers. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[23] Georgios E. Fainekos. Revising Temporal Logic Specifications for Motion Planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 40–45, 2011.

[24] Goldberg, E. and Novikov, Y. Verification of Proofs of Unsatisfiability for CNF Formulas. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 886–891, 2003.

[25] Hadas Kress-Gazit and Georgios E. Fainekos and George J. Pappas. Translating Structured English to Robot Controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[26] Hadas Kress-Gazit and Georgios E. Fainekos and George J. Pappas. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[27] Hadas Kress-Gazit and Tichakorn Wongpiromsarn and Ufuk Topcu. Correct, Reactive Robot Control from Abstraction and Temporal Logic Specifications, Sept. 2011 .

[28] Howie Choset and Kevin M. Lynch and Seth Hutchinson and George A. Kantor and Wolfram Burgard and Lydia E. Kavraki and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[29] Joseph Y. Halpern and Judea Pearl. Causes and Explanations: A Structural-Model Approach - Part II: Explanations. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 27–34, 2001.

[30] Kangjin Kim and Georgios E. Fainekos. Approximate Solutions For the Minimal Revision Problem of Specification Automata. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 265–271, 2012.

[31] Kangjin Kim and Georgios E. Fainekos and Sriram Sankaranarayanan. On the

Revision Problem of Specification Automata. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5171–5176, 2012.

[32] Könighofer, Robert and Hofferek, Georg and Bloem, Roderick. Debugging Unrealizable Specifications with Model-Based Diagnosis. In *International Conference on Hardware and Software: Verification and Testing (HVC)*, pages 29–45, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] LaValle, Steven M. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[34] Leonardo Bobadilla and Oscar Sanchez and Justin Czarnowski and Katrina Gossman and Steven LaValle. Controlling Wild Bodies Using Linear Temporal Logic. In *Robotics: Science and Systems (RSS)*, Los Angeles, CA, USA, June 2011.

[35] Marius Kloetzer and Calin Belta. A Fully Automated Framework for Control of Linear Systems from Temporal Logic Specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[36] Martin Buehler and Karl Iagnemma and Sanjiv Singh, editor. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic, George Air Force Base, Victorville, California, USA*, volume 56 of *Springer Tracts in Advanced Robotics*. Springer, 2009.

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

[38] Vasumathi Raman and Hadas Kress-Gazit. Explaining impossible high-level robot behaviors. *IEEE Transactions on Robotics*, 29(1):94–104, 2013.

[39] Robert Könighofer and Georg Hofferek and Roderick Bloem. Debugging Formal Specifications Using Simple Counterstrategies. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 152–159, 2009.

[40] Roderick Paul Bloem and Alessandro Cimatti and Karin Greimel and Georg Hofferek and Robert Könighofer and Marco Roveri and Viktor Schuppan and Richard Seeber. RATSY - A New Requirements Analysis Tool with Synthesis. In Springer, editor, *Computer Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 425 – 429, 2010.

[41] Sertac Karaman and Emilio Frazzoli. Sampling-Based Motion Planning with De-

terministic μ-Calculus Specifications. In *IEEE Conference on Decision and Control (CDC)*, 2009.

[42] Shlyakhter, I. and Seater, R. and Jackson, D. and Sridharan, M. and Taghdiri, M. Debugging Overconstrained Declarative Models Using Unsatisfiable Cores. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 94–105, 2003.

[43] Tichakorn Wongpiromsarn and Ufuk Topcu and Richard M. Murray. Receding Horizon Control for Temporal Logic Specifications. In *Hybrid Systems: Computation and Control (HSCC)*, pages 101–110, 2010.

[44] Uri Klein and Amir Pnueli. Revisiting Synthesis of GR(1) Specifications. In *Hardware and Software: Verification and Testing - Proceedings of the 6th International Haifa Verification Conference (HVC)*, volume 6504 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2010.

[45] Vasumathi Raman and Cameron Finucane and Hadas Kress-Gazit. Temporal Logic Robot Mission Planning for Slow and Fast Actions. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 251–256, 2012.

[46] Vasumathi Raman and Constantine Lignos and Cameron Finucane and Kenton Lee and Mitch Marcus and Hadas Kress-Gazit. Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language. In *Robotics: Science and Systems (RSS)*, 2013.

[47] Vasumathi Raman and Hadas Kress-Gazit. Analyzing Unsynthesizable Specifications for High-Level Robot Behavior Using LTLMoP. In *Computer Aided Verification (CAV)*, pages 663–668, 2011.

[48] Vasumathi Raman and Hadas Kress-Gazit. Automated Feedback For Unachievable High-Level Robot Behaviors. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5156–5162, 2012.

[49] Vasumathi Raman and Hadas Kress-Gazit. Towards Minimal Explanations of Unsynthesizability for High-Level Robot Behaviors. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[50] Vasumathi Raman and Nir Piterman and Hadas Kress-Gazit. Provably Correct Continuous Control for High-Level Robot Behaviors with Actions of Arbitrary Execution Durations. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[51] Viktor Schuppan. Towards a Notion of Unsatisfiable Cores for LTL. In *Fundamentals of Software Engineering (FSEN)*, pages 129–145, 2009.

[52] Wenchao Li and Lili Dworkin and Sanjit A. Seshia. Mining Assumptions for Synthesis. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 43–50, 2011.

[53] Yonit Kesten and Nir Piterman and Amir Pnueli. Bridging the Gap between Fair Simulation and Trace Inclusion. In *Computer Aided Verification (CAV)*, pages 381–393, 2003.