Incremental Control Synthesis for Robotics in the Presence of Temporal Logic Specifications

Thesis by

Scott C. Livingston

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California, USA

2016

(Defended 23 November 2015)

Dedicated to the memory of my mother,

Diane Gurulé Livingston

Acknowledgments

I have had the joy of being mentored by Richard M. Murray and Joel W. Burdick, both of whom have repeatedly provided me with good ideas, insight, and feedback. Their styles of working are refreshingly distinct. As my official adviser, Richard maintains an unusual (well, perhaps usual at Caltech) multi-disciplinary group that has allowed me to easily and simultaneously pursue such diverse topics as code generation for flight software at the Jet Propulsion Lab and lab automation in synthetic biology. Since I arrived at Caltech in 2010, I have met many interesting members of Richard's group, Joel's group, and others in Control and Dynamical Systems, the CMS Department, and Caltech broadly. Many of these encounters have led to ongoing collaborations, all of which I joyously acknowledge. Within the scope of this dissertation, I want in particular to thank Pavithra Prabhakar and Eric M. Wolff for inspiration and discussion. Ioannis Filippidis provided me with several useful comments about organization and preliminaries.

I also wish to thank Gerard J. Holzmann and Pietro Perona, who served both on my candidacy and thesis committees, and who have provided thorough and useful criticism of my work, as well as good ideas for future work.

Funding was provided in part by the Boeing Corporation; and separately by United Technologies Corporation and IBM, through the industrial cyberphysical systems (iCyPhy) consortium.

Abstract

This thesis presents methods for incrementally constructing controllers in the presence of uncertainty and nonlinear dynamics. The basic setting is motion planning subject to temporal logic specifications. Broadly, two categories of problems are treated. The first is reactive formal synthesis when so-called discrete abstractions are available. The fragment of linear-time temporal logic (LTL) known as GR(1) is used to express assumptions about an adversarial environment and requirements of the controller. Two problems of changes to a specification are posed that concern the two major aspects of GR(1): safety and liveness. Algorithms providing incremental updates to strategies are presented as solutions. In support of these, an annotation of strategies is developed that facilitates repeated modifications. A variety of properties are proven about it, including necessity of existence and sufficiency for a strategy to be winning. The second category of problems considered is non-reactive (open-loop) synthesis in the absence of a discrete abstraction. Instead, the presented stochastic optimization methods directly construct a control input sequence that achieves low cost and satisfies a LTL formula. Several relaxations are considered as heuristics to address the rarity of sampling trajectories that satisfy an LTL formula and demonstrated to improve convergence rates for Dubins car and single-integrators subject to a recurrence task.

Contents

Chapter 1

Introduction

1.1 Advent of verification and formal synthesis for robotics

A relatively new theme of research in control and dynamical systems is the use of temporal logics to precisely express complex specifications. The new notation goes beyond traditional state and input constraints by providing history dependence and the ability to precisely express sophisticated kinds of behavior like liveness (informally, that a condition repeatedly becomes satisfied) and fairness (informally, if something is repeatedly requested, it is eventually served). Such requirements cannot be captured by traditional notation, and thus describing them has historically involved (and in much industrial practice, still involves) the application of English restricted with special keywords like "shall" and given as lists of bounds on parts of the design. Notoriously, requirements represented in this way can have inconsistencies or be incomplete, perhaps owing to the informal language (English). Besides describing requirements, another challenge that traditional notation did not solve is methodology for the implementation. While languages for process and task description have been around for at least 25 years, the implementation itself has until recently only admitted some code generation for low-level feedback control loops. Otherwise, humans have had to manually design compositions of components, to achieve switching among component controllers by manually using conditional expressions in an implementation language

such as C, etc.

New notation is not progress unless there are significant theoretical results or practical tools that use it. For computer-aided verification, there have been in both respects. During the past 30 years, there has been great progress in the characterization of fundamental problems, e.g., determinacy of parity games, in the development data structures and methods for verifying concurrent software systems, e.g., ordered binary decision diagrams and techniques for SAT solving, and in the available tools for verification, e.g., the Spin model checker. A brief summary of the history of verification and formal logic until around the year 2003 can be found in [55].

While methods of verification demonstrate certain properties about a given system, methods of *synthesis* construct new systems that realize desired properties. In the context of theoretical computer science, "synthesis" tends to refer to the construction of finite-memory policies of action selection for finite transition systems (defined in Chapter 2), which are often regarded as modeling concurrent software processes, especially those that are non-terminating [4]. An early formulation of several synthesis problems in terms of predicates on input and output sequences was given in 1962 by Church [14]. However, it is only recently that practically useful algorithms have begun to be proposed and that specification languages admitting tractable decision procedures or good heuristics have been presented. In control theory, the term "synthesis" is less common, but one of the basic themes is exactly that of constructing controllers that cause trajectories of a dynamical system to meet certain constraints and be robust or optimal according to some objective. In contrast to the setting studied in theoretical computer science, these systems may have state spaces that are differentiable manifolds and may have uncountable time domains (so-called continuous-time systems).

Systems exhibiting aspects that include both of these two broad categories of dynamics are known as being "hybrid." There are definitions that subsume all cases, e.g., transition systems that may have uncountable state sets [58] and transitions defined as solutions of a differential equation, or unified hybrid control systems [11] that can jump, have guard sets, and provide many other potential features expected

of hybrid dynamics. Despite the presence of unified or common notation, hybrid systems can exhibit behaviors that are absent in purely discrete or continuously-valued systems. Therefore new methods of analysis for verification and control have been developed for various kinds of hybrid systems [10, 25]. In summary, hybrid systems are especially challenging to control, and many negative results of undecidability or intractability have been proven [9, 35]. Much current research in the theory of hybrid systems is devoted to finding hybrid dynamics and specification languages that are interesting and have tractable control problems or approximations thereof. As well, developing notions of equivalence, reductions, and simulations are important topics of current work toward synthesis of controllers for difficult problems in this area.

At the same time as the aforementioned developments were made, there has been much progress in robotics research for motion planning. The basic problem concerns moving a rigid body from one point to another in a space among obstacles while avoiding collisions. Solving it precisely is known to be intractable [52]. As such, algorithms that provide exact solutions tend to rely on special structure, such as polygons with single-integrator dynamics navigating among a set of other polygons. In the presence of nonlinear dynamics and, in particular, nonholonomic constraints as occurring in car-like models, techniques of geometric control theory have yielded fast planners. While the point-to-point motion planning problem only requires particular trajectories, practical issues like measurement noise and actuation disturbances have motivated methods based on reference trajectory tracking or potential functions. Another broad and important category that has practically been a great success is sampling-based methods, e.g., Probabilistic Road-Map (PRM) or Rapidly-exploring Random Trees (RRT) [36, 38]. Recently, sampling-based point-to-point kinodynamic planners have been proposed that are optimal, in a probabilistic convergence sense [29, 33].

All of these variations of solutions and problems basically involve planning motion from one point to another point. There are potentially many details that make this challenging, such as arising from multi-fingered robotic hands, moving obstacles, constraints on allowed inputs, or uncertainty about pose. However, the ambitions of

the motion are relatively simple when considered as being part of some task or mission. For example, a surveillance robot must repeatedly move around a building, providing a certain amount of coverage, and respond to surprise requests, while periodically stopping at a battery-charging station, subject to hard time constraints because the battery must not become empty before reaching the station. Part of realizing this task involves solving point-to-point motion planning problems. Solutions for the task itself have historically been the product of manual design by humans, sometimes following templates for robot architectures. Algorithms for planning from research in artificial intelligence may be used at more abstract levels, such as A* for navigating a semantic map of the building, e.g., a graph where vertices represent rooms. However, the composition of various planning modules or the relation of outcomes from trajectory generation and tracking to the finite-state machine that guides task completion have historically been manually designed without techniques of formal verification, with which this section began, being applied.

Motivated by more automation while providing guarantees about correctness for task performance and completion, since around 25 years ago [12, 45, 51, 59], and beginning more intensely 9 years ago (e.g., [6, 16]), attention has been devoted to formally verifying robot architectures and synthesizing solutions to tasks. It should be apparent that this essentially recovers problems of verification and synthesis for hybrid systems, but now with an orientation toward systems and tasks that are of particular interest in robotics.

1.2 Topics and contributions of the thesis

Chapter 2 is dedicated to introducing preliminary background material in preparation for the main work of the thesis. Some detail can be found there about terms used here to summarize the topics and contributions.

Despite theoretical progress in methods for synthesis of controllers from specifications involving robotics and hybrid systems in general, practical realizations are rare. This observation is especially salient because experiments and practical demon-

strations are widely valued in the robotics community. Anecdotal interactions by the author of this thesis with people who work on other topics of robotics indicate skepticism of the actual significance of publications about so-called formal methods for robotics. The importance of the ambitions is usually easily recognized. However, there have been very few meaningful physical experiments, and most proposed methods rely on strong assumptions, e.g., control for systems that have trivial single-integrator dynamics, the absence of actuator uncertainty, or perfect knowledge of global position. With enough of these assumptions, it is possible to exactly recover the setting already treated in theoretical computer science, allowing the disappointing occurrence of papers that do little more than re-hash old ideas in a new venue.

Two important features of most problems in robotics are uncertainty and dynamics. These span the (artificial but often made) boundary between perception and control. Uncertainty essentially causes two fundamental problems in mobile robotics: mapping and localization. More broadly, uncertainty manifests from sensor noise, input disturbances, actuator failures, clock drift, lossy communication channels, object detection and tracking, among many other examples. Dynamics can be thought of as a feature of many problems in robotics because motion is always occurring or soon to occur, and the interplay of mechanics with software control is endemic. Obviously these features are not separate, e.g., disparity between a simplified model of dynamics and the physical hardware can be a source of uncertainty.

Addressing uncertainty and dynamics demands limiting or removing assumptions like perfect position information, known and fixed workspaces, and trivial dynamics. It is a premise of the thesis that they are important targets for research in formal methods for robotics. A sketch of the proposed program of research is to revisit previously well-studied cases of uncertainty in robotics, both in terms of specific sources like sensor noise and in terms of specific problems like localization, in the context of formal synthesis. Similarly, methods should specifically address the various major kinds of dynamics that are well known in robotics: differential drive, car-like dynamics, multi-link arms, dexterous manipulation, etc.

In this thesis, methods for synthesis of controllers are presented that address

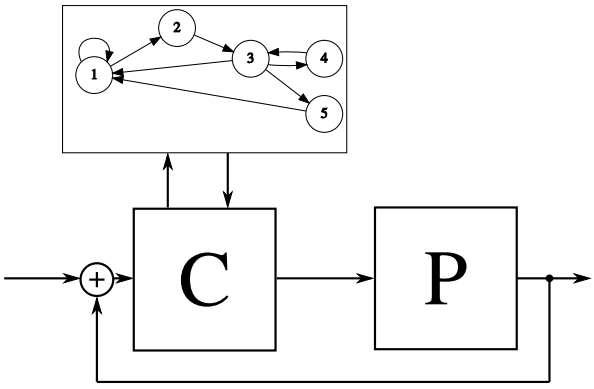

Figure 1.1: Illustration of the form of controllers developed in this thesis, shown in the context of a basic feedback loop. The arrows connecting the boxes are indicative of relationships and the directions of signals. They do not correspond exactly with inputs and outputs.

both of these features. Two problem settings are considered, each involving a broad category of synthesis: reactive and non-reactive (or closed-system). The first setting is essentially formulated as a game for which a finite-memory strategy must be constructed so as to ensure that a specification is satisfied despite an adversarial environment. The specification is written in a fragment of linear-time temporal logic (LTL) referred to as GR(1). The game and strategy are finite structures, i.e., can be regarded as finite directed graphs together with a semantics of execution, turn-taking, and labels of atomic propositions. The problem of synthesis in this setting relies on the existence of a finite representation of a continuously-valued or hybrid dynamical system. Thus the winning strategy that is thought to realize the GR(1) specification is only one part of a controller that also generates trajectories of a plant, as illustrated in Figure 1.1. The relationship between plays in the GR(1) game and trajectories of the plant is made precise using a bisimulation, which is also referred to here as a discrete abstraction to emphasize that one side of the bisimulation equivalence is a transition system with finitely many states.

In the first problem setting, uncertainty as manifested in the GR(1) game is treated. Besides conditions that determine feasible initial states, the two main aspects of specifications are safety and liveness (cf. §2.5). The notion of safety manifests in the game as transition rules, so that from any state, only certain states can be tran-

sitioned to, and the subset of these transitions that are safe depend on the current move of an adversarial environment. The first major contribution of the thesis is to present algorithms for modifying strategies in the presence of changes to transition rules, i.e., changes to reachability in the GR(1) game. The problem and proposed solutions are described as incremental control synthesis because the changes are regarded as occurring sequentially, and so the modifications to strategies so that they are winning with respect to the sequence occur incrementally. As part of the development of solutions in Chapter 3, an annotation for strategies in GR(1) games is presented that is crucial in ensuring that the results of patching are winning (i.e., correct with respect to the modified GR(1) specification). A variety of properties about the annotation is proven, and it is expected to be of interest independent from the incremental synthesis algorithms presented in the thesis.

Part of Chapter 3 is based on [43]. The basic idea of "patching" to recover correctness after a change in workspace reachability was first presented in [42]. The algorithm developed there treats GR(1) synthesis as an opaque subroutine that is invoked to solve restricted specifications based on a subset of the state space and including the changed transition rules, which imply changes to reachability. Solutions of these are modified and then used to patch the original strategy. The methods in Chapter 3 rely on solving a game of lower computational complexity than GR(1) and so are favored to the algorithm of [42]. Though not developed further here, the opaque approach to patching is potentially easily extended to other classes of specifications because the corresponding synthesis algorithms do not need to be revised or decomposed.

The second major contribution of the thesis concerns the other major kind of property expressed by temporal logic specifications: liveness. In Chapter 4, which is based on [41], algorithms are presented for modifying a strategy to be winning after additions to or removals from the sets of game states that must be repeatedly visited. These "sets... to be repeatedly visited" can be thought of as goal sets in a chain of motion planning problems. However, the setting of Chapter 4 is more challenging because it is an adversarial game, and plays are of infinite duration.

The problems posed in Chapters 3 and 4 all begin with a GR(1) formula φ that is modified to yield a new GR(1) formula. The basic practical motivation is that φ is known ahead of time and thus can be solved without significant time constraints. However, details may be missing or the model of motion at the abstract level of the game may have errors. During execution, observations from sensor data or news from a human operator that affects the task can imply modifications to φ. Taken together, the solutions presented in Chapters 3 and 4 are sufficient to handle all substantial respects in which φ (and correspondingly, the game) could thus change, with the exception of changes to assumptions about environment liveness. The only other part of the GR(1) formula template is the initial conditions. Modifications to initial conditions do not affect the play currently in progress (recall the motivation of online changes occurring), and in any case, new initial states are easily handled as solutions to reachability games that are introduced in Chapter 3.

The methods for incremental control synthesis amidst changes to GR(1) formulae address more than uncertainty in task specifications. Recall that the synthesized strategy is interpreted as being part of a controller that drives a plant, possibly involving hybrid dynamics. In practice, the GR(1) game is not only an expression of desired behavior but also constraints among reachable regions, which in turn depend on a cell decomposition of the state space, or on the capabilities for trajectory generation for driving a vehicle among cells. As such, refining the partition can manifest as new transition rules in the GR(1) formula, at which level the presented algorithms are relevant.

Studying reactive synthesis for GR(1) specifications, compared to other specification languages, is well motivated because despite superficially appearing simple, GR(1) is quite expressive [46]. Adding more expressiveness can cause much greater computational complexity [31], and thus it represents a good practical trade-off. Basic synthesis algorithms for GR(1) are amenable to the use of binary decision diagrams (BDDs), a representation for sets of states that can be substantially smaller than an enumerative data structure. By comparison, while other fragments have been introduced with specific motivation for problems in robotics, e.g., [64], it is not obvious

how to leverage BDDs with them.

The second problem setting studied in this thesis is that of non-reactive (or open-loop) trajectory generation for nonlinear systems. Whereas in previous chapters the problem setting requires finite decompositions and the existence of discrete abstractions, Chapter 5 presents methods for stochastic optimization of trajectories of a nonlinear system that satisfy an LTL formula and minimize an objective function, all without construction of an abstraction. Furthermore, the labeling of states (in terms of which satisfaction of LTL formulae is defined) does not need to be expressed as unions of polytopes. The proposed algorithm is based on the cross-entropy method. As with any technique for stochastic optimization, obtaining good convergence performance is a key challenge, and to that end, several relaxations (heuristics) of the basic algorithim are also given. Chapter 5 is based on [44].

1.3 Novelty and related work

The methods of Chapters 3 and 4 treat GR(1) specifications. Nonetheless, the salient features are expected to be available for any μ-calculus specification because the reach annotation (defined in §3.6) essentially arises from counting chains of intermediate subsets of fixed-point computations and could be generalized accordingly. Pursuing this is a topic of future work. With that potential relevance of specification languages in mind, a distinguishing aspect of the solutions proposed in those chapters is that the synthesized controller is changed online, as changes to the specification are given. When the specification encodes workspace reachability, as is common in the application of the techniques of formal synthesis in robotics, tolerance to changes is indeed critical. To the best of the author's knowledge, the first paper addressing incremental control for changing temporal logic specifications is [42] (the author's own work, but which is not presented here). The closest category of problems and solutions treated in prior work is robustness, which broadly aims to synthesize controllers that operate correctly in the presence of some class of uncertainties, or that gracefully degrade in the presence of violated assumptions [7]. The controller is made to be robust and not

modified during execution, so is conservative in the sense that not all disturbances may occur.

The broad notion of incremental or on-the-fly methods for synthesis is certainly not novel, and it also occurs in the context of model checking [5, 22, 24, 56, 60]. However, the interpretation of the terms "incremental" and "on-the-fly" is in reference to construction of the transition system that is to be controlled. Usually a method is called "on-the-fly" if the solution can be obtained without constructing the transition system entirely, whether symbolically (i.e., in a representation in terms of sets of states) or enumeratively (i.e., in a representation in terms of individual states). Solutions for the reachability games introduced in §3.7 could be obtained in an "on-the-fly" manner.

Methods for so-called strategy improvement are relevant insofar as it follows the present theme of incrementally modifying a game strategy. As suggested by "improvement," problems in that area of work concern optimization with respect to an objective. Roughly, the solutions are obtained in steps by beginning with some winning strategy and then modifying particular moves taken under it so that cost is reduced while remaining winning in the original parity game [54] (and references therein). The game itself does not change, and a strategy continues to be winning independently of the cost function.

Analogous to strategy improvement in the literature on theoretical computer science is so-called anytime algorithms in the AI and planning literature. The basic theme of anytime planning algorithms is to quickly present a feasible plan (or trajectory, or path, in the context of robotics) and then to improve it as much as time allows [34, 39]. Again, there is no change to the space of feasible solutions. However, there is some relevance in analogy, e.g., if the cost function is slowly changing during time, an anytime algorithm might be able to track the changes.

For control schemes that involve a discrete abstraction, as relied on in Chapters 3 and 4, there could be some choice in the level of abstraction. Thus it is in principle possible to choose a sufficiently coarse abstraction so that uncertainty in the underlying physical system does not appear at the level of satisfaction of an LTL formula.

Obviously this does not solve the problem but merely translates it to a different level. The abstraction thus provides a crucial interface, and there are methods for refining it as needed based on counter-examples [1]. Such work is relevant insofar as it provides a way to cope with uncertainty by preventing its inclusion in the abstracted system. It can thus also be regarded as complementary to the methods in Chapters 3 and 4.

Recently there has been work in robotics considering formal synthesis for specifications expressed in LTL and GR(1) and in the presence of uncertainty or incremental controller construction [23, 53, 61, 65, 67]. A broad organization of this work is possible according to whether it is reactive (in the sense of a game; cf. §2.5) and where uncertainty occurs: in the dynamical system modeling the robot, in the environment or workspace, or in the temporal logic formulae providing the task specification. Obviously there is some overlap among these, and indeed, a question for research is how to trade-off uncertainty among them, possibly as a design choice using abstraction refinement as suggested above.

The algorithm and relaxations presented in Chapter 5 address optimal control for nonlinear systems subject to LTL specifications. There are only two prior works that are comparable. First, Karaman and Frazzoli described a sampling-based motion planner that operates similarly to RRT* [30] and produces trajectories that satisfy deterministic μ-calculus specifications. Second, Wolff and Murray develop a procedure for expressing and then solving the problem as mixed-integer linear programs [63]. Nonlinear is a broad class of systems, so perhaps unsurprisingly both of those methods and that presented in Chapter 5 have several parameters that can be tuned in particular scenarios. As such, direct comparison is difficult, but an attempt is made in §5.5.3.

Chapter 2

Preliminaries

In this chapter, preliminary developments are made in preparation for the main work of the thesis. Besides fixing notation, this chapter also serves to very briefly provide background material appropriate for a general audience who is broadly familiar with control and dynamical systems but not necessarily with the specific topics prerequisite for this thesis. Additional preparatory notes for a more general audience are given in the appendices.

Before beginning, some basic notation is introduced. The set of real numbers is $\mathbb{R}$ (it is also sometimes referred to as *the real line*), the set of integers is $\mathbb{Z}$, and the set of positive integers is $\mathbb{Z}_+$. The natural numbers begin at 0 (and thus are equal as a set to the nonnegative integers) and are denoted by $\mathbb{N}$. Let A and B be sets. The set of all subsets of A is denoted by 2^A. The set of elements in A but not in B (i.e., the *set difference*) is $A \setminus B$.

A closed interval of the real line is denoted by $[a, b]$, where $a \leq b$ and a and b are known as *endpoints*. An open interval with the same endpoints is denoted by $]a, b[$.

2.1 Formal languages

Let Σ be a finite set. A *finite string of* Σ is a function $\sigma : \{0, 1, \ldots, n\} \to \Sigma$, where $n \in \mathbb{N}$. When there is no ambiguity about the set Σ, we simply refer to σ as a *finite string*. The cardinality of the domain of a string is referred to as its *length*; for example, the length of σ is $n + 1$.

Let $\alpha : \{0, 1, \ldots, n\} \to \Sigma$ and $\beta : \{0, 1, \ldots, m\} \to \Sigma$ be strings. Define an operation, denoted by $\alpha\beta$ (i.e., by juxtaposition) and called *concatenation*, that is defined as the function

$$\alpha\beta(t) = \begin{cases} \alpha(t) & \text{if} \quad 0 \leq t \leq n \\ \beta(t - n - 1) & \text{otherwise,} \end{cases} \tag{2.1}$$

where $0 \leq t \leq n + m + 1$. It is immediate that $\alpha\beta$ is a finite string and has length $n + m + 2$. The set of all finite strings of Σ is denoted by Σ^+. A *language* is a subset of Σ^+.

After adding an empty string to Σ^+ (as an identity element under the concatenation operation, and thereby obtaining the Kleene closure Σ^*), we have the basis for studying regular languages, context-free grammars, etc., which are outside the scope of this preliminaries chapter. Interested readers should begin at [26].

Most of the specifications considered in this thesis are for nonterminating systems, i.e., having trajectories of infinite duration. As such, we are also interested in strings that have countably infinite length. An *infinite string of* Σ is a function $\sigma : \mathbb{N} \to \Sigma$. When there is no ambiguity about the set Σ, we simply refer to σ as an *infinite string*. The set of all infinite strings of Σ is denoted by Σ^ω. When the distinction between infinite or finite string is without consequence, or if a property holds in both cases, we simply write *string*. Concatenation as in equation (2.1) is defined between finite strings and infinite strings, in that order, where they are also referred to as *prefix* and *suffix*, respectively.

Let σ be a string. A *substring* of σ is a string obtained by restricting the domain of σ and shifting indices to begin at 0. Precisely, let I be an interval of the form $[a, \infty[$ or $[a, b]$, where $a, b \in \mathbb{N}$, $0 \leq a \leq b$, and b is less than the length of σ if σ is a finite string. Define $\sigma_I(t) = \sigma(t + a)$, where if $I = [a, b]$ then t can take value at most $b - a$ (in which case, $\sigma_{[a,b]}$ has length $b - a + 1$).

2.2 Linear-time temporal logic

Since its introduction as a formalism for specifying properties of programs by Pnueli in 1977 [49], linear-time temporal logic (LTL)[1] has enjoyed widespread adoption and has been the subject of and used in much research. There are many variants and details that can be considered [20], but only the basic propositional construction is used here. General introductions can be found in [4, 15].

As for any specification language, LTL is defined in two parts. First, the syntax determines the notation by defining the formulae that can possibly be written. Second, the semantics are defined in terms of string containment, thus determining when a sequence can be said to satisfy an LTL formula. Before beginning, we introduce the main device of labelings. Let AP be a set of atomic propositions. For readers who are familiar with programming languages like C++ and Python, atomic propositions can be thought of as variables of type `bool` and are thus able to take one of two values: `true` or `false`. More generally, they are the basic (indivisible, i.e., atomic) units of truth-value. States of a dynamical system are labeled according to whether each atomic proposition in AP is `true` or `false` at that state. The labeling is given or otherwise chosen and thus not usually a part of the control synthesis problem; rather, synthesis is described as a problem about realizing certain sequences of truth-values of atomic propositions.

In this thesis, the set-theoretic style is used, wherein each subset $B \subseteq \mathrm{AP}$ is interpreted as an assignment of `true` and `false` to atomic propositions according to presence in or absence from B, respectively. Explicitly, p is `true` if and only if $p \in B$. Because subsets correspond to assignment of values, they are also referred to as *states*. (Thus, $B \subseteq \mathrm{AP}$ is a state, and 2^{AP} is a set of states.) In terms of variables taking values on finite domains, development in terms only of atomic propositions is without loss of generality because any such variable admits a binary representation; cf. §C.2.2.

The syntax of LTL is described as a context-free grammar. In Backus-Nauer form,

[1] LTL is also known as "linear temporal logic." Here the former is preferred to emphasize that linearity refers to time and not a property of functions on vector spaces as elsewhere in the thesis.

the production rules are

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \tag{2.2}$$

where $p \in \mathrm{AP}$ and φ is a nonterminal. The first three productions provide the familiar propositional (non-temporal) logic, sometimes called Boolean logic. The other productions introduce temporal operators. (The meaning of "temporal" will become apparent with the semantics as defined below.) From these, the following operators are syntactically derived:

- $\texttt{true} \equiv p \vee \neg p$
- $\varphi \wedge \psi \equiv \neg\left(\neg\varphi \vee \neg\psi\right)$
- $\varphi \implies \psi \equiv \neg\varphi \vee \psi$,
- $\varphi \iff \psi \equiv (\varphi \implies \psi) \wedge (\psi \implies \varphi)$,
- $\diamondsuit\, \varphi \equiv \texttt{true}\, \mathcal{U}\, \varphi$,
- $\square\, \varphi \equiv \neg \diamondsuit \neg\varphi$,
- $\texttt{false} \equiv \neg\, \texttt{true}$.

Before proceeding to semantics, notice that among the productions in (2.2) there are no parentheses, i.e., the symbols (and). They are not considered a part of the syntax of LTL, but rather, they are used to emphasize precedence. As is conventional, an expression in parentheses is evaluated before any expression containing it. Parentheses are applied to remove ambiguity about parsing an LTL formula, i.e., ambiguity about the sequence in which productions from (2.2) are applied, which affects the interpretation of the formula because the semantics are defined inductively on grammar productions.

Let φ be an LTL formula (having the syntax defined above). Let σ be an infinite string of 2^{AP}, and let $t \in \mathbb{N}$. The satisfaction of φ by σ beginning at time t, denoted by

$\sigma, t \models \varphi$, is defined inductively on the parse tree of φ (i.e., the grammar productions that yield φ) as follows.

- $\sigma, t \models \texttt{true}$;
- $\sigma, t \models p$ if and only if $p \in \sigma_{[t,\infty[}(0)$ (which is equivalent to $p \in \sigma(t)$);
- $\sigma, t \models \neg\varphi$ if and only if $\sigma, t \models \varphi$ is not true, which is written $\sigma, t \not\models \varphi$;
- $\sigma, t \models \bigcirc \varphi$ if and only if $\sigma, t+1 \models \varphi$;
- $\sigma, t \models \Box \varphi$ if and only if for all $\tau \geq t$, $\sigma, \tau \models \varphi$;
- $\sigma, t \models \Diamond \varphi$ if and only if there exists $\tau \geq t$, $\sigma, \tau \models \varphi$;
- $\sigma, t \models \varphi \mathcal{U} \psi$ if and only if $\sigma, t \models \psi$ or there exists $j > 0$ such that $\sigma, t+j \models \psi$ and for all $0 \leq i < j$, $\sigma, t+i \models \varphi$.

If $\sigma, 0 \models \varphi$ (i.e., $t = 0$), then we simply write $\sigma \models \varphi$. Several of the definitions are redundant but included for clarity, e.g., $\sigma \models \Diamond \varphi$ could be obtained using $\sigma \models \texttt{true} \mathcal{U} \varphi$, following the syntactic derivation given earlier.

2.3 The basic synthesis problem

Having introduced LTL as a language for expressing specifications, we need only introduce a structure that admits some notion of control in order to be able to pose a synthesis problem. A *transition system* is a tuple $T = (S, I, \mathrm{Act}, \rightarrow, L, \mathrm{AP})$ where S is a set of states (not necessarily finite), $I \subseteq S$ is a set of initial states, and $\rightarrow \subseteq S \times \mathrm{Act} \times S$ is a relation. States are labeled with the function $L : S \rightarrow 2^{\mathrm{AP}}$. This sketch of transition systems follows the literature [4, 58]. Precise problems are formulated in later chapters, but it is useful to provide a sketch of a basic synthesis problem here.

Problem 1 (sketch)**.** Let φ be an LTL formula in terms of the atomic propositions AP. Find a partial function $C : S^+ \times \mathbb{N} \rightarrow \mathrm{Act}$ such that all state sequences of the transition system T under C have labeling satisfying φ.

2.4 The modal μ-calculus

A more expressive language than LTL, which was introduced in §2.2, is μ-calculus. In this section it is briefly introduced with a focus on only those parts relevant for this thesis. Applications and research involving μ-calculus are rich, and readers who are generally interested in it should consult [21, 55]. The primary motivation to introduce μ-calculus, despite working primarily with specifications that can be expressed in LTL, is that it readily admits so-called fixed-point algorithms that provide one basis for strategy synthesis. In particular, intermediate values provide sequences of sets of states that may be useful for reachability computations, as in §3.5.

Let AP be a set of atomic propositions, and let Var be a set of variables. A *Kripke structure* is a triple $\mathcal{K} = (S, \mathcal{R}, L)$ where S is a finite set of states, $\mathcal{R} \subseteq S \times S$ is a relation, and $L : S \to 2^{\mathrm{AP}}$ is a function that labels each state with a set of atomic propositions that are `true` in that state. An *execution* of $\mathcal{K}$ is a sequence of states $s : \mathbb{N} \to S$ such that $(s(t), s(t+1)) \in \mathcal{R}$ for all $t \geq 0$. The *trace* of an execution s is the function $w : \mathbb{N} \to 2^{\mathrm{AP}}$ such that $w(t) = L(s(t))$ for $t \geq 0$. An initial state could also be defined as part of the Kripke structure, but it is not needed here. A transition system, as defined in §2.3, in which there are no control actions available is equivalent to a Kripke structure (after defining initial states).

Let $\mathcal{X} \in \mathrm{Var}$ be a variable, and let $Q \subseteq S$ be a set of states of the Kripke structure $\mathcal{K}$. Define the new Kripke structure $\mathcal{K}^Q_{\mathcal{X}} = (S, \mathcal{R}, L')$ over the set of atomic propositions $\mathrm{AP}' = \mathrm{AP} \cup \{\mathcal{X}\}$ and where

$$L'(s) = \begin{cases} L(s) \cup \{\mathcal{X}\} & \text{if } s \in Q \\ L(s) & \text{otherwise.} \end{cases}$$

The syntax of μ-calculus is defined by the following grammar productions.

$$\phi ::= p \mid \mathcal{X} \mid \phi \vee \phi \mid \neg\phi \mid \Diamond \phi \mid \Box \phi \mid \mu\mathcal{X}.\phi \mid \nu\mathcal{X}.\phi$$

Analogously to the relationship of LTL and infinite strings, μ-calculus semantics are

defined on Kripke structures. Observe that a trace of a Kripke structure is just an infinite string of 2^{AP}, so the semantics of LTL can as well be defined on Kripke structures. Proceeding inductively on the syntax, we have

$$\begin{aligned}
[\![p]\!]_{\mathcal{K}} &= \{s \in S \mid p \in L(s)\} \\
[\![\phi \vee \psi]\!]_{\mathcal{K}} &= [\![\phi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}} \\
[\![\neg\phi]\!]_{\mathcal{K}} &= S \setminus [\![\phi]\!]_{\mathcal{K}} \\
[\![\Box\,\phi]\!]_{\mathcal{K}} &= \{s \in S \mid \forall s' \in S.\, (s, s') \in \mathcal{R} \implies s' \in [\![\phi]\!]_{\mathcal{K}}\} \\
[\![\Diamond\,\phi]\!]_{\mathcal{K}} &= \{s \in S \mid \exists s' \in S.\, (s, s') \in \mathcal{R} \wedge s' \in [\![\phi]\!]_{\mathcal{K}}\}
\end{aligned}$$

Consider the recursion on subsets of S defined by

$$\begin{aligned}
Q_0 &= \emptyset \\
Q_{k+1} &= Q_k \cup [\![\phi]\!]_{\mathcal{K}^{Q_k}_{\mathcal{X}}}.
\end{aligned}$$

It turns out that there is some k^* where $Q_{k^*+1} = Q_{k^*}$, i.e., it is a fixed-point under the function defined by $[\![\phi]\!]_{\mathcal{K}^{Q}_{\mathcal{X}}}$. $[\![\mu\mathcal{X}.\,\phi]\!]_{\mathcal{K}}$ is defined to be this set. Similarly, $[\![\nu\mathcal{X}.\,\phi]\!]_{\mathcal{K}}$ is defined inductively by

$$\begin{aligned}
Q_0 &= S \\
Q_{k+1} &= Q_k \cap [\![\phi]\!]_{\mathcal{K}^{Q_k}_{\mathcal{X}}}.
\end{aligned}$$

When writing μ-calculus formulae, the formatting convention will be used where states are lowercase latin letters, e.g., x, sets of states are uppercase, e.g., $X \subseteq S$, and variables are written with a calligraphic style, e.g., $\mathcal{X} \in \mathrm{Var}$. The μ-calculus is not used extensively and thus the convention suffices, especially as a suggestive notation relating the variables with sets of states, the latter being of practical interest.

2.5 Reactivity, games, and another basic synthesis problem

The adjective "reactive" is commonly used in robotics to refer to a style of control that, informally, lacks much foresight or planning. Instead, actions are selected based on superficial interpretations of sensor measurements. A small example is a thresholding routine that stops all motion if any range finder values are too small (and presumably are indicative of potential collisions). The basic paradigm is demonstrated in the robot architectures broadly categorized as being behavior-based [13].

Following instead the convention in the theoretical computer science literature [50, 55], throughout this thesis the term *reactive* is used to indicate the presence of an adversary that selects values for some of the inputs. This is analogous to the notion of *disturbance* in robust control theory [18, 27]. However, using LTL as introduced in §2.2, we can express a time-varying dependence between disturbances (adversarial or uncontrolled inputs) and the states that should be reached by the controller. This situation can be described as a turn-based game, and the objective is to ensure that an LTL formula is satisfied, despite all possible adversarial strategies.

The intuitive sketch above is now made precise. Let AP be a set of atomic propositions, and partition it into $\mathrm{AP}^{\mathrm{env}}$ and $\mathrm{AP}^{\mathrm{sys}}$. The former set contains atomic propositions that take truth-value according to the choice of an adversary, i.e., they are uncontrolled and thus always present the possibility of the worst-case. The latter set, $\mathrm{AP}^{\mathrm{sys}}$, has propositions that we are able to control. Note that the control (assignment of values to) $\mathrm{AP}^{\mathrm{sys}}$ may only be indirect, e.g., after driving the state of the underlying dynamical system into some polytope, as described in §2.6.

The reactive synthesis problem for LTL is as follows. Let φ be an LTL formula in terms of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$. For any infinite string σ^{env} of $2^{\mathrm{AP}^{\mathrm{env}}}$, find an infinite string σ^{sys} of $2^{\mathrm{AP}^{\mathrm{sys}}}$ such that the combination, defined as

$$\sigma(t) = \sigma^{\mathrm{env}}(t) \cup \sigma^{\mathrm{sys}}(t)$$

for $t \geq 0$, and being a string of $2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$, satisfies φ. While this problem formulation is attractive because it is consistent with the perspective of systems as functions of entire input sequences to entire output sequences, there are obvious practical difficulties.

Example 1. Let $\mathrm{AP}^{\mathrm{env}} = \{p\}$, and let $\mathrm{AP}^{\mathrm{sys}} = \{q\}$. The reactive LTL specification

$$q \iff \Box p$$

has only solutions that are not causal. Intuitively, the right-side of the formula is satisfied if for all time p is `true`. However, the left-side is satisfied if q is `true` at the initial state. Thus, this instance of the reactive synthesis problem requires a controller that can predict the value of p. Since the adversary decides the truth-value of p and has no constraints, this is practically impossible.

The reactive synthesis treated in this thesis is for a fragment of LTL known as GR(1) (generalized reactivity of rank 1, or generalized Streett[1]) [31]. Together with some conditions determining initial states (given below), GR(1) is syntactically defined by the formula template

$$\Box \rho^{\mathrm{env}} \wedge \left(\bigwedge_{j=0}^{m-1} \Box \diamondsuit \psi_j^{\mathrm{env}} \right) \implies \Box \rho^{\mathrm{sys}} \wedge \left(\bigwedge_{i=0}^{n-1} \Box \diamondsuit \psi_i^{\mathrm{sys}} \right), \tag{2.3}$$

where each subformula is defined in terms of atomic propositions as follows. All of ψ_j^{env} and ψ_i^{sys} are formulae in terms of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$ and without temporal operators. ρ^{env} is a formula in terms of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$ and can only contain the temporal operator $\bigcirc$ in direct application to atomic propositions of $\mathrm{AP}^{\mathrm{env}}$, i.e., only as subformulae $\bigcirc p$ where $p \in \mathrm{AP}^{\mathrm{env}}$. Finally, ρ^{sys} is a formula in terms of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$ and can only contain the temporal operator $\bigcirc$ in direct application to atomic propositions of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$, i.e., only as subformulae $\bigcirc p$ where $p \in \mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$. The left-side of the implication in (2.3) is commonly referred to as the "assumption," and the right-side as the "guarantee."

A *play* is an infinite sequence of subsets of $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$, i.e., a function $\sigma : \mathbb{N} \to 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$, where at time $t \geq 1$, $\sigma(t)$ is determined in two steps:

1. the environment selects a subset $e \subseteq \mathrm{AP}^{\mathrm{env}}$;

2. given e, the system selects a subset $s \subseteq \mathrm{AP}^{\mathrm{env}}$,

and then $\sigma(t) = e \cup s$. This interpretation of turn-taking is known as a Mealy time semantics (cf. Appendix A). Keeping this in mind, a play σ is simply an infinite string of $\Sigma = 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$, and as such, the notation defined in §2.2 for satisfaction of an LTL formula can be used, e.g., $\sigma \models \varphi$, where φ is of the form (2.3).

A *GR(1) game* is a pair $\mathcal{G} = (\iota, \varphi)$ where φ is of the form (2.3), $\iota \in \Sigma$ is the *initial state*. A play σ is said to be *initial* if $\sigma(0) = \iota$. An *environment strategy* is a function $g : \left(2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}\right)^+ \to 2^{\mathrm{AP}^{\mathrm{env}}}$. A (system) *strategy* is a partial function $f : \left(2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}\right)^+ \times 2^{\mathrm{AP}^{\mathrm{env}}} \to 2^{\mathrm{AP}^{\mathrm{sys}}}$. A play $\sigma : \mathbb{N} \to 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$ is said to be *consistent* with environment strategy g and system strategy f if for all $t \geq 0$,

$$\sigma(t+1) = g(\sigma_{[0,t]}) \cup f(\sigma_{[0,t]}, g(\sigma_{[0,t]})). \tag{2.4}$$

By the Mealy time semantics, the presence of $g(\sigma_{[0,t]})$ as an argument of f is well-defined. A system strategy f is said to be *winning* if and only if for every environment strategy g and for every initial play σ that is consistent with f and g, $\sigma \models \varphi$.

Problem 2 (GR(1) synthesis)**.** Let $\mathcal{G} = (\iota, \varphi)$ be a GR(1) game. Find a system strategy f that is winning, or decide that one does not exist.

If a winning strategy exists, the GR(1) game $\mathcal{G}$ is said to be *realizable*. In this case, a winning strategy f is said to realize $\mathcal{G}$. When the initial state does not need to be distinguished, for brevity we may only refer to a GR(1) game directly as the LTL formula φ.

Because (ι, φ) can be thought of as specifying requirements for a controller given assumptions about its environment, the term GR(1) *specification* may be used instead of game. These terms are interchangeable, although "specification" seems popular in engineering practice.

While Problem 2 is in terms of a single initial state, there are several possibilities for deciding the initial states and, together with φ, thereby defining the set of winning plays. The distinction is not crucial for this thesis. For completeness, three common choices are outlined here, all of which can be represented using the single initial state formulation given above, possibly after an appropriate modification to the transition rules. Let Init $\subseteq 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$.

1. Plays can begin at any state in Init. The (adversarial) environment can arbitrarily select it.

2. Plays begin at some state, which can be chosen along with the system strategy.

3. For each assignment of $\mathrm{AP}^{\mathrm{env}}$ (chosen by the environment), the system strategy must choose an assignment of $\mathrm{AP}^{\mathrm{sys}}$ such that the combined state is in Init.

The definition of the GR(1) synthesis problem relies on the interpretation of plays as infinite strings. However, if a state transition occurs in which ρ^{env} or ρ^{sys} is violated, then the play is decided (winning if the former is violated; not winning otherwise), and the remaining infinite string suffix is not relevant. In particular, a winning strategy could be one that forces the environment to violate ρ^{env}. Such winning strategies will not be addressed in this thesis. Indeed, because the suffix of a play is not relevant, a different problem can be posed as finite-time reachability, which is subsumed by reachability games that are posed and solved in §3.7.

The basic synthesis procedure for GR(1) is crucial for some of the development in §3 and is outlined there in §3.5.

2.6 Finite representations of dynamical systems

One way to bring formal languages to bear on deciding properties about trajectories of hybrid systems is to discretize particular trajectories of the system and then label the sequence of states using a given function, e.g., which demarcates regions of interest like goals and obstacles in the state space. This essentially is the approach

taken in Chapter 5. Alternatively, a finite transition system may be constructed that preserves appropriate aspects of the original system, i.e., the physical system that we actually want to control. The relationship between these two systems is a simulation or bisimulation [2, 4, 40, 58]. There are many possibilities for achieving this, and indeed, construction of and control for abstractions is a topic of current research. Important extensions have also been proposed, including probabilistic and approximate bisimulations.

Basic ingredients are illustrated by the approach taken in [66], where in summary the state space for a piecewise linear system is partitioned into finitely many polytopes that refine labeling of states in terms of atomic propositions. Abstraction construction involves checking, for each pair of cells, whether it is possible to reach one from any state in the other, using a fixed or varying number of states and allowing for disturbances. Because regions are polytopes and the dynamics are linear, it is possible to pose this as feasibility checking in a linear program. The existence of a feasible point implies the existence of a control sequence from one cell to another. The abstract system is then a directed graph with vertices corresponding to polytopes and edges corresponding to the existence of feasible points, i.e., of input sequences from one cell to the other.

Formal synthesis is then on the finite transition system, and the result is guaranteed to yield trajectories on the original system because of the bisimulation. In this setting, a controller has at least two components: the strategy produced for the finite transition system, and the control method that produces trajectories in the physical system corresponding to transitions of the abstract system. A precise definition is not given here because it is not needed. However, note that the methods of Chapters 3 and 4 are applicable to hybrid systems by way of discrete abstractions.

Chapter 3

Patching for Changes in Reachability

3.1 Introduction

A fundamental problem for dynamical systems is determination of the reachable state space. The construction and analysis of controllers essentially involves controlling the reachable states and the manner in which they are reached, e.g., manifesting for linear systems as basic parameters like rise and settling times [3].

The intuitive setting for this chapter is one in which we have already considered the reachable state space and constructed a controller that satisfies an objective in it. A game is being played against the environment and so the controller is, in part, a strategy that is winning in terms of that game. If some of the possible transitions in the game are changed, what can we do with the nominal strategy that we already have? Obviously it is always an option to discard it and construct another strategy de novo, but in some situations we can do better. This sketch is made into a precise problem in this chapter, and a solution for it is developed. Parts of this chapter are based on joint work with Prabhakar [43].

3.2 The game graph of a GR(1) specification

Let φ be a GR(1) formula as in (2.3), where the set of uncontrolled (environment) atomic propositions is $\mathrm{AP}^{\mathrm{env}}$ and the set of controlled (system) atomic propositions is

$\mathrm{AP}^{\mathrm{sys}}$. For conciseness, let $\Sigma = 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$. Throughout the chapter, elements of Σ, i.e., subsets of $\mathrm{AP}^{\mathrm{env}}$ and $\mathrm{AP}^{\mathrm{sys}}$, will be written as lowercase letters such as x. This convention is to emphasize the perspective of assignments of truth-values to atomic propositions as *states*.

Here and in the next chapter, it will be useful to think of plays in terms of infinite-length walks on a graph obtained from the safety subformulae, ρ^{env} and ρ^{sys}, of φ. Before defining the graph obtained from φ, the notation for satisfaction developed for LTL in §2.2 is extended slightly to provide for transition rules. Let $x, y \in \Sigma$. For $\rho \in \{\rho^{\mathrm{env}}, \rho^{\mathrm{sys}}\}$, define $(x, y) \models \rho$ as the predicate: for every infinite string σ of Σ such that $\sigma(0) = x$ and $\sigma(1) = y$, $\sigma \models \rho$. Intuitively, $(x, y) \models \rho$ holds if and only if every infinite string that has first element x and second element y satisfies ρ. This is useful because, as defined in §2.5, the only temporal operator that can appear in ρ is $\bigcirc$, and it can only be in a subformula of the form $\bigcirc p$ for an atomic proposition p. Thus, satisfaction of it by any infinite string can be decided using only the first two elements. Similarly, for a formula without temporal operators θ, define $x \models \theta$ if and only if $\sigma \models \theta$ for every infinite string σ of Σ such that $\sigma(0) = x$. (The restriction of θ lacking temporal operators could be removed, but the generality is not needed.)

Define the graph $G_\varphi = (\Sigma, E_\varphi^{\mathrm{env}}, E_\varphi^{\mathrm{sys}})$, where Σ is as defined earlier, and for $x \in \Sigma$ and $y \subseteq \mathrm{AP}^{\mathrm{env}}$,

$$E_\varphi^{\mathrm{env}}(x) = \{z \subseteq \mathrm{AP}^{\mathrm{env}} \mid (x, z) \models \rho^{\mathrm{env}}\}, \tag{3.1}$$

$$E_\varphi^{\mathrm{sys}}(x, y) = \{z \subseteq \mathrm{AP}^{\mathrm{sys}} \mid (x, y \cup z) \models \rho^{\mathrm{sys}}\}. \tag{3.2}$$

G_φ is a directed graph in that E_φ^{env} and E_φ^{sys} together provide an edge set, namely $(x, y) \in \Sigma \times \Sigma$ is an edge if and only if $y \cap \mathrm{AP}^{\mathrm{env}} \in E_\varphi^{\mathrm{env}}(x)$ and $y \cap \mathrm{AP}^{\mathrm{sys}} \in E_\varphi^{\mathrm{sys}}(x, y \cap \mathrm{AP}^{\mathrm{env}})$. This definition deviates from the usual definition of game graph in which there are controlled and uncontrolled vertices instead of edges as used here. For completeness, a graph of that form is provided in Appendix A. The motivation for expressing the two transition rules through edges is to facilitate studying the relationship with strategies on it and changes to the game.

3.3 Problem statement

Recall the template for GR(1) formulae (2.3) from §2.5,

$$\square\,\rho^{\text{env}} \wedge \left(\bigwedge_{j=0}^{m-1} \square\diamondsuit\,\psi_j^{\text{env}}\right) \implies \square\,\rho^{\text{sys}} \wedge \left(\bigwedge_{i=0}^{n-1} \square\diamondsuit\,\psi_i^{\text{sys}}\right).$$

Problem 3. Let $\varphi_0, \varphi_1, \ldots$ be an infinite sequence of GR(1) formulae that all have the same subformulae ψ_j^{env}, $j = 0, \ldots, m-1$, and ψ_i^{sys}, $i = 0, \ldots, n-1$, but possibly distinct transition rules, i.e., the sequence of GR(1) formulae is characterized by a sequence of pairs of formulae

$$\left(\rho_0^{\text{env}}, \rho_0^{\text{sys}}\right), \left(\rho_1^{\text{env}}, \rho_1^{\text{sys}}\right), \ldots. \tag{3.3}$$

Find a sequence of strategies $f_0, f_1, \ldots$ such that f_k realizes φ_k, the formula having ρ_k^{env} and ρ_k^{sys}.

The statement as given does not address causality, i.e., whether the entire sequence of specifications is known at once or given incrementally. Practically motivated, we consider the latter, but the proposed solution could as well be used for the former.

3.4 Strategy automata

The definition of strategy given for reactive synthesis allows, in general, dependence on the entire history of a play (cf. §2.5). For GR(1), a finite-memory suffices [31]. In this section, the particular form of finite-memory strategy used in this thesis is defined. (Because only finite memory is required, these strategies are also generically referred to as "finite-state machines." To avoid confusion with the many variants of usage for that term, it is avoided here.)

Let φ be a GR(1) formula as in (2.3), and let $\Sigma = 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}$. A *strategy automaton* for φ is a tuple $A = (V, I, \delta, L)$, where V is a finite set (the elements of V are called *nodes*), $I \subseteq V$ is a set of initial nodes, $L : V \to \Sigma$ is a labeling of nodes with

game states, and δ is a function that determines successor nodes in A given inputs from the environment, i.e., valuations represented as subsets of $\mathrm{AP}^{\mathrm{env}}$, so that state transitions are consistent with ρ^{env} and ρ^{sys}. The domain of δ is

$$\bigcup_{v \in V} \{v\} \times E_{\varphi}^{\mathrm{env}}(L(v)),$$

and for every $v \in V$, $e \in E_{\varphi}^{\mathrm{env}}(L(v))$, $L(\delta(v,e)) \cap \mathrm{AP}^{\mathrm{env}} = e$ and $(L(v), L(\delta(v,e))) \models \rho^{\mathrm{sys}}$ (so, $L(\delta(v,e)) \cap \mathrm{AP}^{\mathrm{sys}} \in E_{\varphi}^{\mathrm{sys}}(L(v), e)$). Intuitively, the domain of δ ensures that a transition exists for each possible move by the environment from each state that can occur in A, and the game state labeling the node that is obtained after the transition is required to be consistent, i.e., the uncontrolled part of the state (in $\mathrm{AP}^{\mathrm{env}}$) is the same as that which enabled the transition leading there, and the labels of node and predecessor together are feasible among available system (robot) moves.

Example 2. Let $\mathrm{AP}^{\mathrm{env}} = \{\texttt{door\_open}, \texttt{door\_reached}\}$ and $\mathrm{AP}^{\mathrm{sys}} = \{\texttt{goto\_door}\}$. Consider the GR(1) formula having

$$\psi_0^{\mathrm{env}} = (\texttt{goto\_door} \rightarrow \texttt{door\_reached})$$
$$\rho^{\mathrm{sys}} = (\texttt{door\_open} \rightarrow \bigcirc \texttt{goto\_door}) \wedge ((\texttt{goto\_door} \wedge \neg \texttt{door\_reached}) \rightarrow \bigcirc \texttt{goto\_door})$$
$$\psi_0^{\mathrm{sys}} = \texttt{door\_reached},$$

which encodes the task of going to a door whenever it becomes open. (The left-side of each equality corresponds with a subformula of (2.3).) If the door is detected as open (as in the left subformula of ρ^{sys}), then the robot transitions into the mode of `goto_door`. It cannot leave that mode until the door is reached, which is indicated by the atomic proposition `door_reached`. The environment can declare when it is reached and when it is open. The only liveness condition assumed to be satisfied by the environment, $\square \diamondsuit (\texttt{goto\_door} \rightarrow \texttt{door\_reached})$, can be thought of as providing a fairness assumption. If the robot continues to try to go to the door, eventually it will be reached.

A strategy automaton realizing the specification is shown in Figure 3.1.

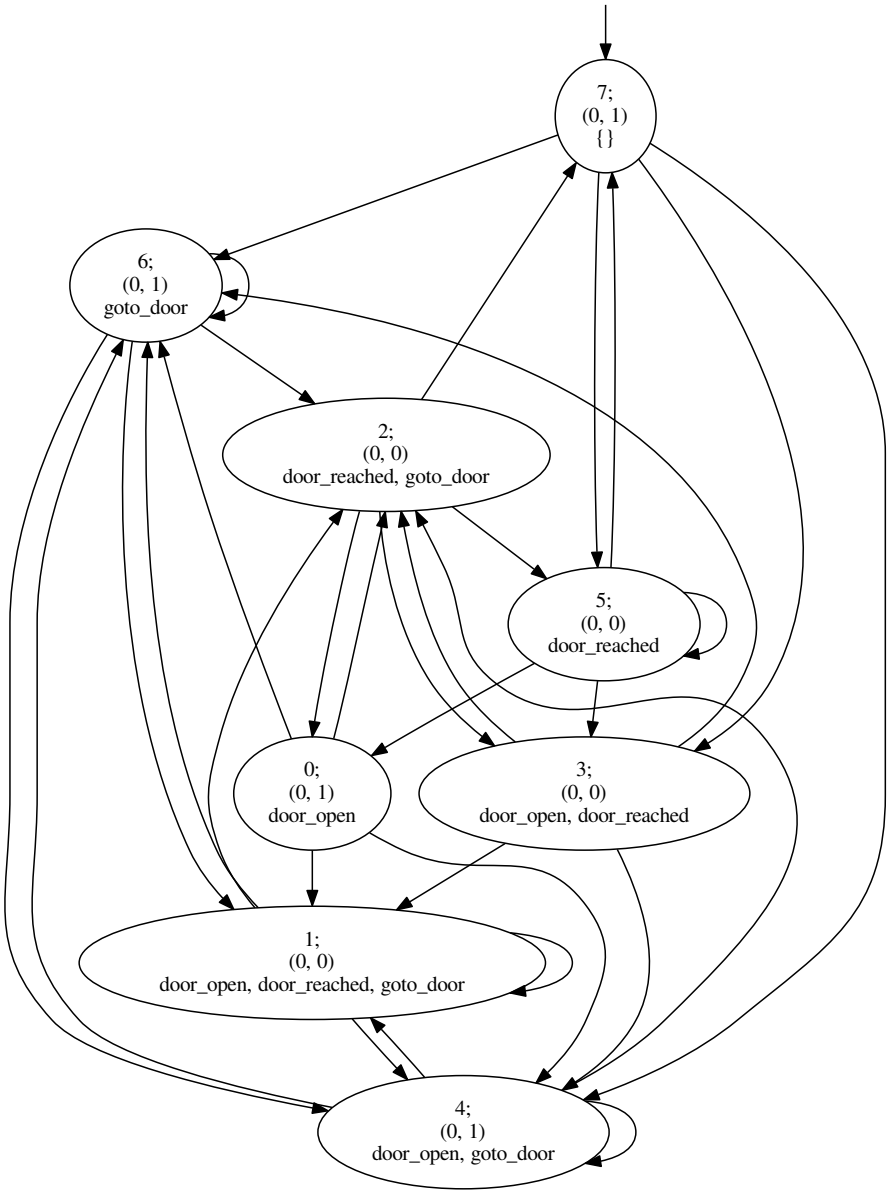

Figure 3.1: A winning strategy automaton for the GR(1) game of Example 2. Each node has three rows. The first row contains an integer that uniquely identifies the node. The second row is a pair of values that is part of a reach annotation for the strategy (defined later in §3.6). Third is the set of atomic propositions that are `true` when execution reaches that node; this corresponds to the L-value of the node. The initial node is indicated by the arrow with no predecessor (shown near the top of the figure). An expression of the specification that can be passed to the tool `gr1c` is given in Appendix C.

In §2.5, GR(1) games are defined as having a single initial state. Therefore, it suffices to always consider a strategy automaton with a singleton set of initial nodes, $I = \{v_0\}$. As discussed there, other interpretations of initial conditions can be recovered with appropriate modification of the set of states Σ or the transition rules ρ^{env} and ρ^{sys}. Other initial conditions could be treated by having more initial nodes in the strategy automaton, without having to construct an equivalent game with one initial state.

Let $A = (V, I, \delta, L)$ be a strategy automaton. The *graph associated with* A has edge set $E(A)$ determined by enumerating possible adversarial inputs from each node, i.e.,

$$E(A) = \{(u, v) \in V \times V \mid \exists x \subseteq \text{AP}^{\text{env}} \,.\, \delta(u, x) = v\}\,. \tag{3.4}$$

As a directed graph $(V, E(A))$ the usual notation can be applied. For any node $v \in V$, the set of successors of v is denoted by $\text{Post}(v) = \{u \in V \mid (v, u) \in E(A)\}$, and similarly the set of predecessors of v is denoted by $\text{Pre}(v)$. An *execution* of A is a function $r : \mathbb{N} \to V$ such that $(r(t), r(t+1)) \in E(A)$ for $t \geq 0$, and $r(0) \in I$. The *trace* associated with the execution r is the function $\mathcal{L}(r) : \mathbb{N} \to \Sigma$ defined by $\mathcal{L}(r)(t) = L(r(t))$ for $t \geq 0$. The restriction of the domain of an execution to a bounded interval is called a *finite execution*. Since V is finite, an execution is just an infinite string, in the terminology of §2.1. Thus, a finite execution is a finite substring.

Notice that a trace of A is an infinite string of Σ, i.e., an infinite sequence of assignments of values to the atomic propositions $\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}$. Thus a trace can be said to satisfy or not the GR(1) formula φ, i.e., whether $\mathcal{L}(r) \models \varphi$ for an execution r. The strategy automaton A is said to be *winning* for a GR(1) game (ι, φ) if $\iota = L(v)$, where $I = \{v\}$, and every trace of A satisfies φ. To be justified, this definition of winning should constructively imply the existence of a winning strategy in the form given for GR(1) synthesis (Problem 2). While this follows from finite-memory being sufficient and the synthesis procedure outlined in §3.5, an explicit construction is given here in preparation for later results. Before doing so, observe that the definition of δ requires that transitions are safe. If the environment takes a move not

in $E_\varphi^{\text{env}}(L(v))$ during an execution that reached node v, then δ is not defined. However, the play is immediately winning, so the strategy automaton can be ignored. Furthermore, from the definition of strategy automaton, it is not possible to reach a state where $E_\varphi^{\text{sys}}(L(v), e)$ is empty, i.e., where there are no safe system (robot) moves from the node v given the permissible environment move e. Such a strategy could not be winning because there would be at least one play in which the environment (adversary) drives the game to an unsafe state. Besides these reasons, encoding safe transitions directly into the definition is well motivated because it aligns with the parity game perspective, in which the safety (transition) formulae ρ^{env} and ρ^{sys} are represented instead as edges in a game graph (cf. §3.2 and Appendix A).

Definition 1. Let (ι, φ) be a GR(1) game, and let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton for it. Let $\bowtie$ be an arbitrary object that is not a node of A (i.e., $\bowtie \notin V$). For any $g : \mathbb{N} \to 2^{\text{AP}^{\text{env}}}$, define the function $\hat{r}^g : \mathbb{N} \to V \cup \{\bowtie\}$ inductively as

$$\hat{r}^g(0) = \begin{cases} v_0 & \text{if } \iota = L(v_0) \wedge g(0) = \iota \cap \text{AP}^{\text{env}} \\ \bowtie & \text{otherwise} \end{cases}$$

$$\hat{r}^g(t+1) = \begin{cases} \delta(\hat{r}^g(t), g(t)) & \text{if } \hat{r}^g(t) \in V \wedge g(t) \in E_\varphi^{\text{env}}(L(\hat{r}^g(t))) \\ \bowtie & \text{otherwise} \end{cases}$$

for $t \in \mathbb{N}$. Equivalently, define the function $\hat{r} : \left(2^{\text{AP}^{\text{env}}}\right)^+ \to V \cup \{\bowtie\}$ as $\hat{r}(g_{[0,t]}) = \hat{r}^g(t)$. Finally, denote the projection of a string σ of $2^{\text{AP}^{\text{sys}} \cup \text{AP}^{\text{env}}}$ onto AP^{env} by

$$\sigma^{\text{env}}(t) = \sigma(t) \cap \text{AP}^{\text{env}}$$

for $t \geq 0$. Then, the *strategy induced by* A is the function $f_A : \left(2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}\right)^+ \times 2^{\text{AP}^{\text{env}}} \to 2^{\text{AP}^{\text{sys}}}$ such that for $\sigma \in \left(2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}\right)^+$ and $e \in 2^{\text{AP}^{\text{env}}}$,

$$f_A(\sigma, e) = \begin{cases} L(\hat{r}(\sigma^{\text{env}} e)) \cap \text{AP}^{\text{sys}} & \text{if } \hat{r}(\sigma^{\text{env}} e) \neq \bowtie \\ \emptyset & \text{otherwise.} \end{cases} \tag{3.5}$$

Using this definition, the next theorem asserts that a winning strategy automaton

yields a strategy winning in the GR(1) game. Thus, it is enough to verify that a given strategy automaton is winning in order to solve a GR(1) game. (This result motivates the repeated use of "winning.")

Theorem 1. *Let A be a strategy automaton that is winning for a GR(1) game (ι, φ). Then the strategy induced by A is winning.*

Proof. Let $A = (V, \{v_0\}, \delta, L)$ be a winning strategy automaton for the GR(1) game (ι, φ). Let f_A be the strategy induced by A. Let $g : \left(2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}\right)^+ \to 2^{\mathrm{AP}^{\mathrm{env}}}$ be an environment strategy. The initial play σ consistent with f_A and g is defined inductively by

$$\sigma(0) = \iota$$
$$\sigma(t+1) = g(\sigma_{[0,t]}) \cup f(\sigma_{[0,t]}, g(\sigma_{[0,t]}))$$

for $t \geq 0$. In Definition 1, $\hat{r}$ is defined on any finite string of $2^{\mathrm{AP}^{\mathrm{env}}}$. Thus, from equation (3.5), f_A is defined on any finite string of $2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$ and subset of $\mathrm{AP}^{\mathrm{env}}$. Denoting projection of σ onto $\mathrm{AP}^{\mathrm{env}}$ by σ^{env} as in Definition 1, we have that $\hat{r}(\sigma^{\mathrm{env}}_{[0,\tau]}) = \bowtie$ if and only if there is a positive $T \leq \tau$ such that $\hat{r}^{\sigma^{\mathrm{env}}}(T) = \bowtie$ and $\hat{r}^{\sigma^{\mathrm{env}}}(t) \in V$ for $t < T$. (Observe that $\sigma(0) = \iota$ and $L(v_0) = \iota$ by hypothesis, hence $\hat{r}^{\sigma^{\mathrm{env}}}(0) \neq \bowtie$ and $T > 0$.) Therefore, $\sigma(T-1) \cap \mathrm{AP}^{\mathrm{env}} \notin E^{\mathrm{env}}_{\varphi}(L(\hat{r}^{\sigma^{\mathrm{env}}}(T-1)))$, i.e., the environment move does not satisfy ρ^{env} by definition of $E^{\mathrm{env}}_{\varphi}$. Because $\hat{r}^{\sigma^{\mathrm{env}}}(t) \in V$ for $t < T$, $\left(L(\hat{r}^{\sigma^{\mathrm{env}}}(t)), L(\hat{r}^{\sigma^{\mathrm{env}}}(t+1))\right) \models \rho^{\mathrm{sys}}$ for $t < T-1$ by the definition of transitions (δ) in strategy automata. Thus, the play satisfies φ. For the other case, i.e., $\hat{r}(\sigma^{\mathrm{env}}_{[0,\tau]}) \neq \bowtie$ for all $\tau \geq 0$, $\hat{r}^{\sigma^{\mathrm{env}}}$ is an execution of A, so $\sigma = \mathcal{L}(\hat{r}^{\sigma^{\mathrm{env}}})$ is a trace of A. By hypothesis A is winning, hence $\sigma \models \varphi$. □

The definition of strategy automaton is distinct from the edge-oriented definition sometimes used for Mealy machines. The motivation for labeling nodes with game states will become apparent when an annotation is introduced in the next section.

3.5 Synthesis for GR(1) as a fixed-point computation

The basic method for synthesis outlined in this section is not a contribution of this thesis, e.g., it has been described earlier in [31] (and later as [8]), albeit with different notation. Nonetheless it plays a crucial role in the remainder of this chapter, so we are motivated to discuss it.

Before beginning, the setting is informally sketched. Intuitively, one manner of constructing a strategy that realizes (2.3) is to pursue states that satisfy ψ_0^{sys} and from which it is possible to reach states satisfying each of the other liveness subformulae, $\psi_1^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$. When this is not possible, there must be some way to block liveness of the environment, i.e., to eventually begin an infinite sequence of states in which ψ_j^{env} is not satisfied, for some $j \in \{0, \ldots, m-1\}$. Provided environment liveness, upon reaching a state that satisfies ψ_0^{sys}, attention shifts to pursuing a state that satisfies ψ_1^{sys}, and the process continues. Because this superficially resembles a sequence of reachability problems on a finite graph, one may guess that a solution is obtained by repeated predecessor computations as familiar for shortest paths problems. However, an important difficulty is that each transition depends on the environment. In other words, the reactivity in this process is the presence of an adversary that, at each time step, can select an arbitrary valuation for part of the state. Thus, the solution is a strategy (not merely a walk on a graph), and the predecessor sets must quantify over possible moves by the environment.

The above sketch is now made precise. Let φ be a GR(1) formula as in (2.3), and recall the definition of the associated graph G_φ given in §3.2. Let $X \subseteq \Sigma$. The set of controlled predecessors of X is defined by

$$\text{Pre}_\varphi(X) = \left\{ y \in \Sigma \mid \forall z_1 \in E_\varphi^{\text{env}}(y).\, \exists z_2 \in E_\varphi^{\text{sys}}(y, z_1).\, z_1 \cup z_2 \in X \right\}. \tag{3.6}$$

The *winning set* is the subset of Σ from which there exist winning system strategies. For the GR(1) formula, this set is called W_φ, and as proven in [31], it can be

found using μ-calculus formulae

$$\nu\mathcal{Z}_i.\left(\mu\mathcal{Y}.\left(\bigvee_{j=0}^{m-1}\nu\mathcal{X}_j.\left((\psi_i^{\text{sys}}\wedge\text{Pre}_\varphi(\mathcal{Z}_{i+1}))\vee\text{Pre}_\varphi(\mathcal{Y})\vee\left(\neg\psi_j^{\text{env}}\wedge\text{Pre}_\varphi(\mathcal{X}_j)\right)\right)\right)\right), \tag{3.7}$$

where the subscript addition of $\mathcal{Z}_{i+1}$ is modulo n (the number of system liveness subformulae in (2.3)), and where $i\in\{0,1,\ldots,n-1\}$. That is, the entire formula $\mathcal{W}_\varphi$ is a chain of subformulae, one (3.7) for each value of i. As outlined in §2.4, $W_\varphi=[\![\mathcal{W}_\varphi]\!]$ is obtained using a fixed-point computation. Moreover, at the fixed-point, $W_\varphi=Z_0=[\![\mathcal{Z}_0]\!]=Z_1=\cdots=Z_{n-1}$, and $W_\varphi=\text{Pre}_\varphi(W_\varphi)$.

Let $i\in\{0,1,\ldots,n-1\}$. An intermediate value of the fixed-point computation used to obtain W_φ is a finite sequence of sets

$$Y_i^0\subset Y_i^1\subset\cdots\subset Y_i^k=W_\varphi, \tag{3.8}$$

for some k, where Y_i^0 is a set of states in W_φ that satisfies ψ_i^{sys}. Furthermore, for each $l\in\{1,2,\ldots,k\}$, from every state $x\in Y_i^l$, for all $z\in E_\varphi^{\text{env}}(x)$, at least one of the following holds:

1. there exists $y\in E_\varphi^{\text{sys}}(x,z)$ such that $z\cup y\in Y_i^{l-1}$, or
2. there is a strategy blocking one of the environment liveness conditions, ψ_j^{env}, that remains within Y_i^l.

Details about construction are given in [8, 31].

3.6 Annotating strategies

To facilitate adaptation (incremental synthesis) to a repeatedly changing task formula, in this section a novel annotation for strategies is introduced and several important properties of it are proven. The basic idea is that some extra information can be saved during synthesis of a nominal strategy and used later to guide local modifications

(patches). Note that the following definition was initially given in [41] and is modified from the earlier version of [43].

Let φ be a GR(1) formula as in (2.3). A state $x \in \Sigma$ is said to be an i-system goal if $x \models \psi_i^{\text{sys}}$. Recall that $\Sigma = 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}$.

Definition 2. A *reach annotation* on a strategy automaton $A = (V, I, \delta, L)$ for a GR(1) formula φ is a function $\text{RA} : V \to \{0, \ldots, n-1\} \times \mathbb{N}$ that satisfies the following conditions. Write $\text{RA}(v) = (\text{RA}_1(v), \text{RA}_2(v))$.

1. For each $v \in V$, $\text{RA}_2(v) = 0$ if and only if $L(v)$ is a $\text{RA}_1(v)$-system goal.

2. For each $v \in V$ and $u \in \text{Post}(v)$, if $\text{RA}_2(v) \neq 0$, then $\text{RA}_1(v) = \text{RA}_1(u)$ and $\text{RA}_2(v) \geq \text{RA}_2(u)$.

3. For any finite execution $v_1, v_2, \ldots, v_K$ of A such that $\text{RA}_2(v_1) = \cdots = \text{RA}_2(v_K) > 0$, there exists an environment goal ψ_j^{env} such that for all $k \in \{1, \ldots, K\}$, $L(v_k)$ does not satisfy ψ_j^{env}.

4. For each $v \in V$ and $u \in \text{Post}(v)$, if $\text{RA}_2(v) = 0$, then there exists a p such that for all r between $\text{RA}_1(v)$ and p, $L(v)$ is a r-system goal, and $\text{RA}_1(u) = p$. Specifically, if $p < \text{RA}_1(v)$, the numbers between p and $\text{RA}_1(v)$ are $p+1, \ldots, \text{RA}_1(v)-1$, and if $\text{RA}_1(v) \leq p$, then the numbers between p and $\text{RA}_1(v)$ are $p+1, \ldots, n-1, 0, \ldots, \text{RA}_1(v)-1$.

An illustration of a strategy automaton with a reach annotation on it is given in Figure 3.2. It realizes a deterministic (without adversarial environment) game. The strategy automaton shown earlier in Figure 3.1 as part of Example 2 provides a slightly more complicated demonstration of a reach annotation.

Remark 2. During basic synthesis, a reach annotation can be constructed using the indices of the intermediate values (Y_i^l in (3.8)) of the fixed-point computation on (3.7) and thus does not affect asymptotic complexity of the basic GR(1) synthesis algorithm. (It affects the multiplicative constant.)

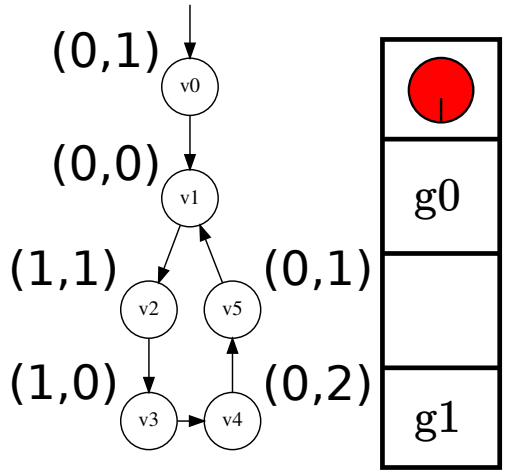

Figure 3.2: Illustration of a deterministic (without adversarial environment) specification, a strategy automaton that realizes it, and a reach annotation. The task is expressed as $\square\diamondsuit \texttt{g0} \wedge \square\diamondsuit \texttt{g1}$.

In other words, we can obtain an initial reach annotation on a nominal winning strategy at no extra (asymptotic) cost. This is significant because global synthesis is already difficult, scaling exponentially with the number of atomic propositions $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$.

Theorem 3. *Let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton for the GR(1) game (ι, φ), where $L(v_0) = \iota$. If* RA *is a reach annotation on A for φ, then A is winning.*

Proof. Let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton for the GR(1) game (ι, φ), where $L(v_0) = \iota$, and let RA be a reach annotation on A. Let $r : \mathbb{N} \to V$ be an execution of A. Writing $\mathrm{RA}(v) = (\mathrm{RA}_1(v), \mathrm{RA}_2(v))$ for each $v \in V$, two infinite sequences are obtained from function composition with the execution: $\mathrm{RA}_1 \circ r$ and $\mathrm{RA}_2 \circ r$. The former is a function from $\mathbb{N}$ to the finite set $\{0, \ldots, n-1\}$ and thus presents two cases. First, suppose there is some $K \geq 0$ such that for all $t \geq K$, $\mathrm{RA}_1 \circ r(t) = \mathrm{RA}_1 \circ r(K)$, i.e., $\mathrm{RA}_1 \circ r$ is eventually constant. From Definition 2, this can happen only if at least one of the following occurs. If $\mathrm{RA}_2 \circ r(t) = 0$ for all $t \geq K$, then $\mathcal{L}(r)(t)$ satisfies $\psi_0^{\mathrm{sys}} \wedge \cdots \wedge \psi_{n-1}^{\mathrm{sys}}$, i.e., every state reached after time t is an i-system

goal, for all i. Otherwise (possibly in addition to the previous), there is some $T \geq K$ such that $\mathrm{RA}_2 \circ r(t) = \mathrm{RA}_2 \circ r(T) > 0$ for $t \geq T$. This follows from the monotonicity of RA_2 when RA_1 is not changing, and by definition, one of ψ_j^{env} is not satisfied for all $t \geq T$. Therefore if $\mathrm{RA}_1 \circ r$ is eventually constant, the corresponding trace satisfies φ. Second, suppose that for every $t \geq 0$, there is a $\tau > t$ such that $\mathrm{RA}_1 \circ r(t) \neq \mathrm{RA}_1 \circ r(\tau)$, i.e., $\mathrm{RA}_1 \circ r$ is not eventually constant. From Definition 2, for every $t \geq 0$ where $\mathrm{RA}_1 \circ r(t) \neq \mathrm{RA}_1 \circ r(t+1)$, $\mathrm{RA}_2 \circ r(t) = 0$, which by the definition implies that $L(r(t))$ is a $\mathrm{RA}_1 \circ r(t)$-system goal. The definition also requires that $L(r(t))$ is a p-system goal for all p between $\mathrm{RA}_1 \circ r(t)$ and $\mathrm{RA}_1 \circ r(t+1)$. Since this range of numbers is strictly increasing modulo n, it follows that all of $\psi_0^{\mathrm{sys}}, \ldots, \psi_{n-1}^{\mathrm{sys}}$ are satisfied infinitely often. Therefore $\mathcal{L}(r) \models \varphi$. Because the execution was arbitrary and every trace is from some execution, therefore A is a winning strategy automaton. □

Recall from Theorem 1 that a winning strategy automaton indeed wins the GR(1) game. Taken together with Theorem 3, to verify that a strategy automaton realizes a GR(1) specification, it is enough to present a reach annotation on it. This observation will be used later to demonstrate that modifications yield new winning strategies, thereby solving Problem 3.

Theorem 4. *There exists a winning strategy automaton for a GR(1) game (ι, φ) if and only if there exists a winning strategy automaton with a reach annotation for φ.*

Proof. The converse is trivial. For the other direction, suppose there exists a winning strategy automaton $A = (V, \{v_0\}, \delta, L)$, where $L(v_0) = \iota$. The proof proceeds by constructing a new strategy automaton that is winning and by presenting a reach annotation for it. Let $\psi_0^{\mathrm{sys}}, \psi_1^{\mathrm{sys}}, \ldots, \psi_{n-1}^{\mathrm{sys}}$ be the system liveness subformulae in the GR(1) formula φ (cf. (2.3)). Define the tuple $\hat{A} = (\hat{V}, \{(v_0, 0)\}, \hat{\delta}, \hat{L})$, where $\hat{V} = V \times \{0, \ldots, n-1\}$, $\hat{L}((v,i)) = L(v)$ for all $(v,i) \in \hat{V}$, and

$$\hat{\delta}((v,i),e) = \begin{cases} (\delta(v,e), i+1 \mod n) & \text{if } L(v) \models \varphi_i^{\mathrm{sys}} \\ (\delta(v,e), i) & \text{otherwise} \end{cases}$$

for all $(v,i) \in \hat{V}$, for all $e \in E^{\text{env}}_{\varphi}(\hat{L}((v,i)))$. Clearly $\hat{A}$ is a strategy automaton. It is also winning, because there is a bijection between executions, and hence traces, of $\hat{A}$ and A. Now define $\text{RA} : \hat{V} \to \{0,\ldots,n-1\} \times \mathbb{N}$ as follows. For each $(v,i) \in \hat{V}$, $\text{RA}_1((v,i)) = i$, and

$$\text{RA}_2((v,i)) = \begin{cases} 0 & \text{if } L(v) \models \varphi^{\text{sys}}_i \\ 1 & \text{otherwise.} \end{cases}$$

Combining these as $\text{RA}((v,i)) = (\text{RA}_1((v,i)), \text{RA}_2((v,i)))$, it follows that RA is a reach annotation on $\hat{A}$. Furthermore, $\hat{L}((v_0,0)) = L(v_0) = \iota$. By Theorem 3, $\hat{A}$ is winning. □

The construction of a new strategy together with reach annotation in the proof of Theorem 4 implies the following.

Corollary 5. *Given a winning strategy automaton $A = (V, \{v_0\}, \delta, L)$, a winning strategy automaton A' (possibly equal to A) together with a reach annotation on A' can be constructed in time $O(n(|V| + |E(A)|))$, where $E(A)$ is defined by equation* (3.4).

3.7 Reachability games

The final preparation before introducing algorithms that solve Problem 3 is to pose a game that can be regarded as a restriction of GR(1) games. The synthesis problem posed here is smaller (easier to solve, in a precise sense) than GR(1). A solution procedure is described, and strategies for it are shown to admit an annotation similar to that introduced in Definition 2. These strategies are used later as modifications (patches) to a given strategy automaton.

As in previous sections, let $\Sigma = 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}$, which is referred to as the set of (game) states. For a set of states $Q \subseteq \Sigma$, the *characteristic formula* is the Boolean (non-temporal) formula χ_Q that has support of (is satisfied precisely on) Q, i.e., for each $x \in \Sigma$, x satisfies χ_Q if and only if $x \in Q$.

Let $Q, F \subseteq \Sigma$ be sets of states, and let φ be a GR(1) formula (cf. (2.3) in §2.5). The *reachability game* from Q to F, denoted by $\text{Reach}_{\varphi}(Q,F)$, is the reactive LTL

formula

$$\chi_Q \wedge \square\, \rho^{\text{env}} \wedge \left(\bigwedge_{j=0}^{m-1} \square \diamondsuit\, \psi_j^{\text{env}} \right) \implies \square\, \rho^{\text{sys}} \wedge \diamondsuit\, \chi_F, \tag{3.9}$$

for which the semantics are extended slightly from those introduced in §2.2 to allow satisfaction in finite time. For a finite string $\alpha : [0, T] \to \Sigma$, $\alpha \models \text{Reach}_\varphi(Q, F)$ if and only if $\alpha(T) \in F$ and $\alpha\sigma \models \text{Reach}_\varphi(Q, F)$ for some infinite string $\sigma : \mathbb{N} \to \Sigma$. (Recall that $\alpha\sigma$ is the result of concatenation, as defined in §2.1, and thus is itself an infinite string of Σ.)

Using the same time semantics as for GR(1) and further allowing plays to be finite, a game is obtained from $\text{Reach}_\varphi(Q, F)$. A system strategy is a partial function $f : \Sigma^+ \times 2^{\text{AP}^{\text{env}}} \to 2^{\text{AP}^{\text{sys}}}$, and an environment strategy is a partial function $g : \Sigma^+ \to 2^{\text{AP}^{\text{env}}}$. A string σ (possibly finite) is *consistent with* strategies f and g if

$$\sigma(t+1) = g(\sigma_{[0,t]}) \cup f(\sigma_{[0,t]}, g_{[0,t]}),$$

which is just (2.4) of §2.5. (Consistent strings may also be called *plays*, following terminology for GR(1) games.) A system strategy f is said to be *winning for* $\text{Reach}_\varphi(Q, F)$ if, for every environment strategy g and for every σ that is consistent with f and g, $\sigma \models \text{Reach}_\varphi(Q, F)$.

The problem of synthesis (i.e., finding a strategy that is winning) for $\text{Reach}_\varphi(Q, F)$ will be referred to as a *reachability game*. A method for synthesis is now given as a μ-calculus formula based on that used for solving GR(1) games. Let $F \subseteq \Sigma$, and define

$$\text{Local}_\varphi(F) = \mu\mathcal{Y}.\, \left(\bigvee_{j=0}^{m-1} \nu\mathcal{X}.\, \left(\chi_F \vee \text{Pre}_\varphi(\mathcal{Y}) \vee \left(\neg\psi_j^{\text{env}} \wedge \text{Pre}_\varphi(\mathcal{X}) \right) \right) \right), \tag{3.10}$$

which is obtained by removing the outermost fixed-point operators, $\nu\mathcal{Z}_i$, in (3.7) and replacing $(\psi_i^{\text{sys}} \wedge \text{Pre}_\varphi(\mathcal{Z}_{i+1}))$ with χ_F. As such, the solution procedure is entirely similar except for reaching states in F, in which case the strategy can terminate, i.e., reach a node without outgoing transitions.

Solution strategies are defined as follows. A *reach strategy automaton* for the

reachability game $\text{Reach}_\varphi(Q, F)$ is $A = (V, I, \delta, L)$, where V and L are defined as for strategy automata (cf. §3.4), I is a set of initial nodes such that, for each $x \in Q$, there is precisely one $v \in I$ such that $L(v) = x$, and δ is defined as for strategy automata except that a node v is not necessarily in the domain of δ (i.e., v can have no outgoing transitions) if $L(v) \in F$, in which case v is called a *terminal node*. An *execution* r of A is a string (possibly finite) of V such that $r(0) \in I$, for each $t \geq 0$, there is some $e \in E_\varphi^{\text{env}}(L(r(t)))$ such that $r(t+1) = \delta(r(t), e)$, and r is finite only if $L(r(T)) \in F$, where r has length $T + 1$.

A *partial reach annotation* RA on a reach strategy automaton A for a reachability game $\text{Reach}_\varphi(Q, F)$ is a function $\text{RA} : V \to \mathbb{N}$ that satisfies the following conditions:

1. For each $v \in V$, $\text{RA}(v) = 0$ if and only if $L(v) \in F$.

2. For each $(u, v) \in E(A)$, if $\text{RA}(u) \neq 0$, then $\text{RA}(u) \geq \text{RA}(v)$, where $E(A)$ is defined by equation (3.4).

3. For any finite execution $v_1, v_2, \ldots, v_K$ of A such that $\text{RA}(v_1) = \cdots = \text{RA}(v_K) > 0$, there exists an environment goal ψ_j^{env} such that for all $k \in \{1, \ldots, K\}$, $L(v_k)$ does not satisfy ψ_j^{env}.

Comparing with Definition 2, a partial reach annotation is entirely similar to a reach annotation as defined for GR(1) games if $n = 1$ (i.e., if there is one system liveness requirement). Accordingly, proofs for the following are entirely similar to those given in §3.6, with the extra details of beginning at every state in Q and of treating finite plays that occur when a terminal node is reached.

Theorem 6. *Let $A = (V, I, \delta, L)$ be a reach strategy automaton for the reachability game $\text{Reach}_\varphi(Q, F)$. If RA is a partial reach annotation on A for $\text{Reach}_\varphi(Q, F)$, then A is winning.*

Theorem 7. *There exists a winning reach strategy automaton for a reachability game $\text{Reach}_\varphi(Q, F)$ if and only if there exists a winning reach strategy automaton with a reach annotation for $\text{Reach}_\varphi(Q, F)$.*

Remark 8. A partial reach annotation can be constructed using the indices of the intermediate values (Y_i^l in (3.8)) of the fixed-point computation on (3.10) and thus does not affect asymptotic complexity of the basic reachability game synthesis algorithm.

For brevity and when the meaning is clear from context, reach strategy automata are also called *substrategies*.

3.8 Game changes that affect a strategy

Let (ι, φ) be a GR(1) game, let A be a strategy automaton that is winning for it, and let RA be a reach annotation on A. Recall from Theorem 4 that being realizable implies that we can find a winning strategy automaton with a reach annotation. For Problem 3, consider the second GR(1) formula φ' that occurs (φ being the first in the sequence). It is possible that A is winning for φ' without modification. For example, if φ describes assumptions and requirements for an entire building, yet a controller realizing it is able to keep the robot in a single room (and be correct), then changes to φ that affect assumptions about a different room can be ignored.

Recall the game graph G_φ associated with φ from §3.2. The change to a new GR(1) formula φ' can be interpreted as a modification to the edge set G_φ. There are two conditions in which the strategy automaton needs to be modified in order to be winning for φ'.

1. New moves are available for the environment at one of the states that can be reached in a play consistent with A, i.e.,

$$\mathrm{Cond}_1(u) = E^{\mathrm{env}}_{\varphi'}(L(u)) \setminus E^{\mathrm{env}}_{\varphi}(L(u)) \neq \emptyset. \tag{3.11}$$

2. One of the control (system) actions that may be taken by the strategy automaton A is no longer safe, i.e.,

$$\mathrm{Cond}_2(u) = \exists e \in E^{\mathrm{env}}_{\varphi}(L(u)) : \delta(u, e) \notin E^{\mathrm{sys}}_{\varphi'}(L(u), e). \tag{3.12}$$

Both of these are predicates on the set of nodes in the strategy automaton. They are used below to concisely enumerate nodes that are affected by the change to the GR(1) formula φ. Let n be the number of system liveness (goal) subformulae $\psi_0^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$ in φ and φ'; these do not change in the sequence of GR(1) formulae presented in Problem 3. For each $i \in \{0, \ldots, n-1\}$, define

$$U_i = \{v \in V \mid (\text{RA}_1(v) = i) \wedge (\text{RA}_2(v) \neq 0) \wedge (\text{Cond}_1(v) \vee \text{Cond}_2(v))\}, \qquad (3.13)$$

which is called the set of *affected nodes for goal mode i*. Recall from Definition 2 that $\text{RA}_2(v) = 0$ implies that $L(v)$ satisfies $\psi^{\text{sys}}_{\text{RA}_1(v)}$, for $v \in V$. The definition of U_i can thus be equivalently expressed in terms of satisfying ψ_i^{sys}.

Observe that there are only two other ways in which $G_{\varphi'}$ can differ from G_φ. First, $E^{\text{env}}_\varphi(L(u)) \setminus E^{\text{env}}_{\varphi'}(L(u)) \neq \emptyset$ for some $u \in V$. But then from the state $L(u)$, the environment has fewer moves available in the GR(1) game φ' than in φ, and thus from that state the strategy automaton A is still correct (but conservative). Second, for some $u \in V$ and some $e \in E^{\text{env}}_\varphi(L(u))$, $E^{\text{sys}}_{\varphi'}(L(u), e) \setminus E^{\text{sys}}_\varphi(L(u), e) \neq \emptyset$. But then from the state $L(u)$ and given the environment move e from there, the system has more moves available in the GR(1) game φ' than in φ, so again the strategy automaton is still correct. Therefore, the two predicates Cond_1 and Cond_2 defined above fully characterize the effect of changing the GR(1) formula as in Problem 3 on a given strategy automaton.

Let $\text{Affected} = \{u \in V \mid \text{Cond}_1(u) \vee \text{Cond}_2(u)\}$, and observe that $\bigcup_{i=0}^{n-1} U_i$ is not necessarily equal to Affected (but is obviously a subset). The problem is now solved by decomposing it into two cases. First, an algorithm is presented for the case of $\bigcup_{i=0}^{n-1} U_i = \text{Affected}$, i.e., when all of the affected nodes are between goal states, in §3.9. Second, an algorithm is presented for the other case in §3.10.

3.9 Algorithm for patching between goal states

3.9.1 Overview

For the first algorithm, it is assumed that for each $i \in \{0, \ldots, n-1\}$ where $U_i \neq \emptyset$, the nodes v in the strategy automaton A where $\mathrm{RA}(v) = (i, 0)$, i.e., where $L(v)$ is an i-system goal state, are not affected. Under this assumption, the basic effort in modifying A is to find a new way to reach one of these surviving i-system goal states. For this, a reachability game is solved and then patched into A. An algorithm for the other case is given in the sequel, §3.10.

The main steps of the algorithm are outlined as follows. A subset of states $N \subseteq \Sigma$ is given; it should be sufficiently large to contain all states that label nodes in $\cup_i U_i$. In practice this can be a norm ball of game states around a physical position at which the requirements or assumptions about reachability have changed. In summary, for each i with nonempty U_i:

1. Create the set N_i of nodes of A that have label in N and goal mode i (i.e., RA_1 value of i).

2. Create the set Entry^i of labels of nodes in N_i that can be entered from outside N_i in the original strategy. Also include in this set the current state of the play (assuming the algorithm is being used online, we must address how to move from the current position).

3. Compute the minimum RA_2 value (related to the reach annotation yielded by the automaton) over all nodes in $U_i \cup \left(L^{-1}(\mathrm{Entry}^i) \cap \mathrm{RA}_1^{-1}(i)\right)$, and call it m^*.

4. Create the set Exit^i of states of nodes v in N_i with $\mathrm{RA}_2(v)$ less than m^*. Precisely,
$$\mathrm{Exit}^i = \{L(v) \,|\, v \in N_i, \mathrm{RA}_2(v) < m^*\}.$$

5. Compute a winning strategy A^i for the reachability game $\mathrm{Reach}_{\varphi'}(\mathrm{Entry}^i, \mathrm{Exit}^i)$. If it cannot be realized, then fail (cf. discussion below about completeness of

the algorithm). Otherwise, use it to mend the original strategy A by replacing nodes in N_i with the nodes in A^i, and adding appropriate edges corresponding to the edges into the entry states and the edges from the exit nodes in A.

Intuitively, the construction of the sets Entry and Exit is guided by the values of the reach annotation to ensure that, if a local strategy is found from Entry to Exit, then the reach annotation values must have decreased. The monotonicity provides for correctness of executions in the modified strategy automaton A', in particular avoiding the creation of a cycle that provides a trace of A' that does not satisfy φ'.

Finally, after having applied patches to obtain a strategy automaton A', a new reach annotation is constructed for it. This is performed by scaling some of the existing RA_2 values of the given reach annotation on A, and then using the annotations from the solutions of the reachability games after adding an offset value depending on the lowest node from the original A at which the patch is attached. Thus presenting a reach annotation RA' on A' for φ', correctness follows from Theorem 3, and the same algorithm can be applied again on A' and RA', etc.

3.9.2 Formal statement

A precise statement is presented in Algorithms 1 and 2. Several clarifications are the following:

- $N \subseteq \Sigma$ is a precondition asserting that a set of neighborhood states is given. Note that N can be defined by a predicate on $\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}$.

- Line 11: Plays can begin at the initial node v_0, which corresponds to the state ι. Thus, if v_0 is in N_i, ι must be included in Entry^i independent of predecessors of v_0.

- Line 14 of Algorithm 1: The reachability game $\mathrm{Reach}_{\varphi'}(\mathrm{Entry}^i, \mathrm{Exit}^i)$ is solved from a fixed-point computation on the μ-calculus formula $\mathrm{Local}_{\varphi'}(\mathrm{Exit}^i)$, as defined in (3.10). It is described as local because the controlled predecessor

Algorithm 1 Find local strategies

INPUT: GR(1) formula φ, strategy automaton $A = (V, \{v_0\}, \delta, L)$, reach annotation RA, modified formula φ', neighborhood $N \subseteq \Sigma$
OUTPUT: set of triples $(A^i, \text{Entry}^i, \text{Exit}^i)$ with partial reach annotation RA^i
`Patches` $:= \emptyset$
find affected node sets $U_0, U_1, \ldots, U_{n-1}$ //equation (3.13).
for all i such that $U_i \neq \emptyset$ **do**
  $N_i := \{v \in V \mid L(v) \in N \wedge \text{RA}_1(v) = i\}$
  if $U_i \setminus N_i \neq \emptyset$ **then**
    error — N_i is too small.
  end if
  $\text{Entry}^i := \{L(v) \mid v \in N_i \wedge \exists u \in V \setminus N_i : (u, v) \in E(A)\}$
  if $v_0 \in N_i$, then $\text{Entry}^i := \text{Entry}^i \cup \{L(v_0)\}$
  $m^* := \min_{v \in U_i \cup \left(L^{-1}(\text{Entry}^i) \cap \text{RA}_1^{-1}(i)\right)} \text{RA}_2(v)$
  $\text{Exit}^i := \{L(v) \mid v \in N_i \wedge \text{RA}_2(v) < m^* \wedge \exists u \in V \setminus N_i : (v, u) \in E(A)\}$
  if $\text{Entry}^i \not\subseteq [\![\text{Local}_{\varphi'}(\text{Exit}^i)]\!]$ **then**
    error — local problem unrealizable.
  else
    synthesize A^i for $\text{Reach}_{\varphi'}(\text{Entry}^i, \text{Exit}^i)$ with partial reach annotation RA^i
    `Patches` $:=$ `Patches` $\cup \left\{\left(\left(A^i, \text{Entry}^i, \text{Exit}^i\right), N_i, \text{RA}^i\right)\right\}$
  end if
end for
return `Patches`

operation $\text{Pre}_{\varphi'}$ can be constrained to N, thereby reducing the size of the set of vertices reachable in the game graph.

- Line 19 of Algorithm 2: Find the minimum positive integer such that the distance between the Entry and Exit sets (i.e., $m^* - m_*$) is greater than the maximum partial reach annotation value m_{local} on the local strategy A^i.

3.10 Algorithm for patching across goal states

A limitation of the algorithm presented in the previous section is that nodes in A corresponding to goal satisfaction cannot be removed. When some of the goals $\psi_0^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$ depend on the position of the robot, it can occur that a wandering obstacle transiently covers goal states that would be reached under A, or that a new static obstacle that intersects a goal region is discovered. An ambition of this

Algorithm 2 Merge local strategies into the original

INPUT: GR(1) formula φ, strategy automaton $A = (V, \{v_0\}, \delta, L)$, reach annotation RA, modified formula φ', neighborhood $N \subseteq \Sigma$, and `Patches` from Algorithm 1.
OUTPUT: Strategy automaton A' for φ' with reach annotation RA$'$
for all $\big((A^i = (V^i, I^i, \delta^i, L^i), \text{Entry}^i, \text{Exit}^i), N_i, \text{RA}^i\big) \in$ `Patches` **do**
  for all $s \in \text{Entry}^i$ **do**
    find $u \in I^i$ such that $L^i(u) = s$
    for all $v \in V \setminus N_i$ such that $(v, u') \in E(A)$ for some $u' \in N_i$ with $L(u') = s$ **do**
      replace edge (v, u') of δ by (v, u)
    end for
  end for
  $m_* := \max_{v \in L^{-1}(\text{Exit}^i) \cap \text{RA}_1^{-1}(i)} \text{RA}_2(v)$
  for all $u \in V^i$ such that $\text{RA}^i(u) = 0$ (equivalently, $L^i(u) \in \text{Exit}^i$) **do**
    let $v \in N_i$ such that $\text{RA}_2(v) < m^*$ and $L(v) = L^i(u)$
    change all outgoing edges from v to be outgoing from u
  end for
  $m_{local} := \max_{v \in V^i} \text{RA}^i(v)$
  for all $v \in V^i$ **do**
    $\text{RA}^i(v) := \text{RA}^i(v) + m_*$
  end for
  $\alpha := \min_{k \in \mathbb{Z}_+, k(m^* - m_*) > m_{local}} k$
  for all $v \in V$ with $\text{RA}_1(v) = i$ **do**
    $\text{RA}_2(v) := \alpha \cdot \text{RA}_2(v)$
  end for
  $V := (V \setminus N_i) \cup V^i$
  remove edges from δ which intersect with N_i
  add all edges from δ^i to δ
  extend L to include nodes in V^i
  $A' := (V, \{v_0\}, \delta, L)$ and RA$'$:= RA.
end for

section is to achieve robustness against more types of uncertain moving obstacles. Furthermore, the development here provides a certain monotonicity with respect to approaching global re-synthesis as the portion of A that is mended increases.

For any node $v \in V$, the first element $\text{RA}_1(v)$ is said to be the *goal mode* of v. Intuitively, upon reaching the node v, automaton A is pursuing a state that satisfies $\psi^{\text{sys}}_{\text{RA}_1(v)}$. Whether it is actually making progress toward that goal is indicated by the second element RA_2. If RA_2 decreases across an edge (u, v) in A (i.e., $\text{RA}_2(v) <$

$\mathrm{RA}_2(u)$), then strict progress is made. Otherwise one of the environment liveness conditions $\psi_0^{\mathrm{env}}, \ldots, \psi_{m-1}^{\mathrm{env}}$ in (2.3) is not being satisfied. When the goal ψ_i^{sys} of the current mode i being pursued is reached, the goal mode is incremented (modulo n) until an index j is found such that ψ_j^{sys} is not satisfied at the current state. Such a transition corresponds to an edge $(u, v) \in E(A)$ where $\mathrm{RA}_1(u) = i$ and $\mathrm{RA}_1(v) = j$.

In order to study the structure of plays under A in terms of reaching goal states, first partition the set of nodes according to goal mode, i.e.,

$$V_i = \{v \in V \mid \mathrm{RA}_1(v) = i\}, \tag{3.14}$$

and then define an edge set over this partition by

$$\hat{E} = \{(V_i, V_j) \mid i \neq j \wedge \exists u \in V_i, \exists v \in V_j : (u, v) \in E\}. \tag{3.15}$$

Note that $\hat{E}$ is well-defined because $V_0, V_1, \ldots, V_{n-1}$ are mutually disjoint. From Definition 2, it should be clear that edges in $\hat{E}$ correspond to satisfaction of task goals $\psi_0^{\mathrm{sys}}, \ldots, \psi_{n-1}^{\mathrm{sys}}$ because across any such edge, the mode RA_1 changes.

Consider the directed graph $\left(\{V_0, V_1, \ldots, V_{n-1}\}, \hat{E}\right)$, which is obtained from (2.3) and a winning strategy automaton equipped with reach annotation RA as described previously. It turns out that this graph is a chain for many tasks in robotics, as is now shown. By construction of RA, if the goal mode changes by more than 1, then more than one goal must be satisfied upon reaching that node during any play. Precisely this is expressed as

Remark 9. If there exists an edge $(u, v) \in E$ such that

$$|\mathrm{RA}_1(v) - \mathrm{RA}_1(u) \mod n| > 1,$$

then there exist goal indices i, j such that $[\![\psi_i^{\mathrm{sys}}]\!] \cap [\![\psi_j^{\mathrm{sys}}]\!] \neq \emptyset$.

Typically the goals in a robot task are disjoint. For a small example, consider surveillance of several locations in an office building. The robot cannot simultaneously occupy multiple locations at once, and therefore the corresponding goal con-

ditions $\psi_0^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$ in (2.3) are disjoint. This remains true even if variables not corresponding to physical positions are introduced into the task formula φ, provided that position variables are combined with the other variables by conjunction, e.g., go to that room and capture a photograph. In Boolean logic notation, this property is expressed by

$$\psi_i^{\text{sys}} \wedge \psi_j^{\text{sys}} \equiv \texttt{false} \qquad \text{for all } i \neq j, \tag{3.16}$$

or in terms of sets of states, $[\![\psi_i^{\text{sys}}]\!] \cap [\![\psi_j^{\text{sys}}]\!] = \emptyset$ for all distinct pairs of indices i, j.

Therefore, for any robot task satisfying (3.16), it follows from Remark 9 that the graph $\left(\{V_0, V_1, \ldots, V_{n-1}\}, \hat{E}\right)$ can only have edges of the form (V_i, V_{i+1}) or (V_i, V_{i-1}), where index arithmetic is modulo n. Thus, any cycle either has length 2 or n. (It is immediate from the definition of $\hat{E}$ in (3.15) that there cannot be self-loops, i.e., cycles of length 1.) Without loss of generality, we can assume that system goals are pursued in order, as appearing in the task formula φ. To justify this, notice that mode i only concerns pursuit of a particular goal ψ_i^{sys}—satisfaction of a different goal ψ_j^{sys}, $j \neq i$, en route will not affect the current mode. Combining this with previous observations, the following lemma is proven.

Lemma 10. *The graph* $\left(\{V_0, V_1, \ldots, V_{n-1}\}, \hat{E}\right)$ *is a subgraph of a chain.*

Note that the lemma states "subgraph of a chain" because a winning strategy automaton can block one of the environment liveness conditions ψ_j^{env} of (2.3), possibly causing a node V_i to have no outgoing edges. A practical example is a strategy in which the robot blocks the doorway, thus preventing the door from closing, as may have been assumed to always eventually happen.

The extension to the patching algorithm of §3.9 can now be presented. Let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton that is winning for the GR(1) game (ι, φ). Let $N \subseteq \Sigma$ be a set of discrete states, e.g., corresponding to a ball in the robot workspace, over which the strategy from A must be changed. A change is required because a new task formula φ' was obtained due to modification of the transition rules, i.e., one step in an instance of Problem 3. As in the previous section, using the inverse of the node labelling function L, we can find sets of nodes that must be replaced and partition

them according to goal mode. For each $i = 0, 1, \ldots, n-1$, define

$$N_i = \{u \in V \mid L(u) \in N \wedge \mathrm{RA}_1(u) = i\}.$$

Clearly, $N_i \subseteq V_i$, and indeed, this set is equivalently expressed by $N_i = L^{-1}(N) \cap V_i$. Regarding the sets $N_0, N_1, \ldots, N_{n-1}$ as vertices in a graph, a new edge set $\hat{E}_N$ is constructed similarly to $\hat{E}$ in (3.15). Since each N_i is a subset of V_i, an edge is in $\hat{E}_N$ only if it is in $\hat{E}$. Therefore it follows from Lemma 10 that the graph $\left(\{N_0, N_1, \ldots, N_{n-1}\}, \hat{E}_N\right)$ is also a subgraph of a chain.

There are three possibilities for each part of this graph. We treat them separately and summarize the steps for modifying the automaton A to recover correctness with respect to the modified task formula φ' by changing behavior over the states in N. A formal statement is given in Algorithm 3. Before examining the three possibilities for $\left(\{N_0, N_1, \ldots, N_{n-1}\}, \hat{E}_N\right)$, observe that any edge in $\hat{E}_N$ corresponds to a robot goal being reached, i.e., (N_i, N_{i+1}) corresponds to an action by the automaton A in which a ψ_i^{sys}-state is visited and the goal mode is incremented, modulo n. The first possibility is that this graph is itself a chain; this is equivalently stated as the case where there is transition into a goal state for each of the goals. Motivated by practical rarity, in this case we simply perform global re-synthesis of the strategy automaton A for the modified task formula φ'.

For the remaining two possibilities, the graph $\left(\{N_0, N_1, \ldots, N_{n-1}\}, \hat{E}_N\right)$ is not itself a chain. However, recall that it is a subgraph of a chain, and therefore each of the sets N_i is either isolated (i.e., without ingoing or outgoing edges) or part of a finite sequence of edges of the form $(N_i, N_{i+1}), \ldots, (N_{i+\kappa-1}, N_{i+\kappa})$. An example of this is given in Figure 3.3. In the former case, the nodes of A in N_i can be replaced without changing the goal mode i. As such, the algorithm of the previous section can be applied.

In the latter case, let I be a subset of $\{0, 1, \ldots, n-1\}$ indexing the nodes N_i that form a finite sequence of edges in $\hat{E}_N$. We create a new substrategy by solving a sequence of reachability games. To begin, let the initial index in I be i, so that

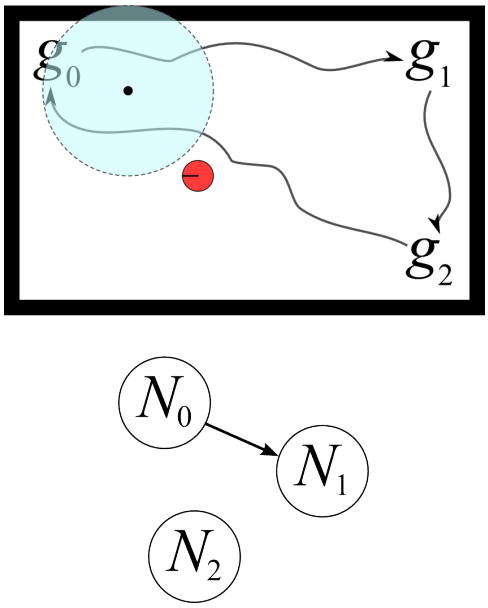

Figure 3.3: Example of the graph $\left(\{N_0, N_1, N_2\}, \hat{E}_N\right)$ of sets of affected nodes, over which patching will occur. The workspace is shown above this graph, with a light blue circle indicating states over which re-planning should occur. In this case, there are three system goals: $\psi_0^{\text{sys}}, \psi_1^{\text{sys}}, \psi_2^{\text{sys}}$, which correspond to positions g_0, g_1, and g_2, respectively, in the workspace. Note that the curves indicate possible paths, whereas A is actually a strategy and thus could lead to many different paths. Nodes in A that correspond to pursuit of ψ_2^{sys}-states are not affected by the change set N in this example. Thus, $N_2 = \emptyset$, and the N_2 vertex is isolated in the above figure. By contrast, there is an edge from N_0 to N_1, indicating that a goal state is reached by an edge affected by the change set N. Thus, in terms of Algorithm 3, the index set I_0 will have two elements, and the substrategy corresponding to that edge will reach a ψ_0^{sys}-state en route to the Exit set.

$I = \{i, i+1, \ldots, i+(|I|-1)\}$, where index arithmetic is modulo n. The sets of nodes Entry^I and Exit^I are created in a manner similar to that in the previous section, except that $\text{Entry}^I \subseteq L(N_i)$ and $\text{Exit}^I \subseteq L(N_{i+(|I|-1)})$. Beginning at the end of this sequence, first solve $\text{Reach}_{\varphi'}([\![\psi^{\text{sys}}_{i+(|I|-1)-1}]\!], \text{Exit}^I)$. This method terminates either when $[\![\psi^{\text{sys}}_{i+(|I|-1)-1}]\!]$ is obtained, or when a fixed-point occurs. In the former case, let $B_{i+(|I|-1)-1} := [\![\psi^{\text{sys}}_{i+(|I|-1)-1}]\!]$, and in the latter case, let $B_{i+(|I|-1)-1}$ be the intersection of the fixed-point with $[\![\psi^{\text{sys}}_{i+(|I|-1)-1}]\!]$. Note that, having been obtained from $\text{Reach}_{\varphi'}()$, it is possible to reach Exit^I from any initial state in $B_{i+(|I|-1)-1}$. Now another reachability game is solved: $\text{Reach}_{\varphi'}([\![\psi^{\text{sys}}_{i+(|I|-1)-2}]\!], B_{i+(|I|-1)-1})$. As in the previous step, store the result to $B_{i+(|I|-1)-2}$. This process is repeated until B_i

is computed. If $\text{Entry}^I \subseteq B_i$, then strategies for each of the reachability games are chained together so that the resulting strategy automaton can drive the actual game, which is governed by transition rules in φ', from any initial state in Entry^I to some state in Exit^I, while visiting robot goals $\psi_i^{\text{sys}}, \psi_{i+1}^{\text{sys}}, \ldots, \psi_{i-(|I|-1)-1}^{\text{sys}}$ en route. Otherwise, the algorithm aborts with failure.

In terms of Algorithm 3, if substrategy construction succeeds for each element I_κ of the partition of indices, then the resulting substrategies in `Patches` can be patched into the original automaton A' in an entirely similar manner as for singleton I_κ in the algorithm of §3.9.

3.11 Analysis

Several remarks and a theorem concerning correctness of the combination of Algorithms 1 and 2 are given in this section. Analysis of the method of Section 3.10 is not presented because it is quite similar to that of Chapter 4, as well as that given below.

Remark 11. For each $i \in \{0, \ldots, n-1\}$, Exit^i (cf. definition on Line 13 of Algorithm 1) does not share any states with Entry^i or $L(U_i)$. Precisely,

$$\text{Exit}^i \cap \big(\text{Entry}^i \cup L(U_i)\big) = \emptyset.$$

Recall from §3.2 that a graph $G_\varphi = (\Sigma, E_\varphi^{\text{env}}, E_\varphi^{\text{sys}})$ associated with the GR(1) game of φ can be constructed. It is equivalent in that the synthesis problem can be posed as controlling transitions on G_φ and ensuring that sets of vertices are repeatedly visited (system liveness satisfaction) or persistently avoided (environment liveness violation). In the analysis below, changes to ρ^{env} and ρ^{sys} are considered, which is equivalent to changes in the edge functions E_φ^{env}, E_φ^{sys}.

Suppose that $N \subseteq \Sigma$ is chosen such that the patching problem is feasible, i.e., the algorithm terminates producing a modified strategy automaton $A' = (V', \{v_0\}, \delta', L')$.

Theorem 12. *The output of Algorithm 2, A' and RA', is such that A' is a strategy automaton for φ' and RA' is a reach annotation on A' with respect to φ'. Therefore, A' is a winning strategy automaton for (ι, φ').*

Proof. To show that $A' = (V', \{v_0\}, \delta', L')$ is a strategy automaton with initial node labeled as ι, proof of the existence of an infinite execution is given by induction on the graph structure of A', such that any dead-end implies the play would be automatically winning because ρ^{env} is not satisfied by the move of the environment (other player). The constructed execution is called r, and corresponds to a play from the labeling $\mathcal{L}(r)$.

The base case is $r(0) = \iota$. First, observe that there are two possibilities concerning the initial node v_0 of A. First, if it is in N_i for some i, then ι is in Entry^i (cf. Line 11 of Algorithm 1). For there to have been some A' and RA' returned, the algorithms must have returned without error. Thus, the reachability game involving ι as labeling v_0 must have been solved, and therefore there is some node in A' that is labeled with ι and which we again call v_0. (If $v_0 \in N_i$, then it is deleted at Line 23 of Algorithm 2, so it can instead be identified with the replacement v_0 that is from V^i.) Let $e \in 2^{\mathrm{AP}^{\mathrm{env}}}$. If $e \notin E^{\mathrm{env}}_{\varphi'}(\iota)$, then any suffix to the play is winning for the system. Otherwise, since v_0 has successors according to the reach strategy automaton that solves $\mathrm{Reach}_{\varphi'}(\mathrm{Entry}^i, \mathrm{Exit}^i)$, there must be some $u \in V'$ such that $(v_0, u) \in E(A')$, $L'(u) \cap \mathrm{AP}^{\mathrm{env}} = e$, and $L'(u) \cap \mathrm{AP}^{\mathrm{sys}} \in E^{\mathrm{sys}}_{\varphi'}(\iota, e)$. Take $r(1) = u$. The second possibility is that v_0 is not in N_i for any i, hence that v_0 is not in U_i for any i. It follows from the hypothesis that A is a strategy automaton for φ and that v_0 must have successors in A' that are labeled with the same states as in A that there is some $u \in V'$ such that $(v_0, u) \in E(A')$, $L'(u) \cap \mathrm{AP}^{\mathrm{env}} = e$, and $L'(u) \cap \mathrm{AP}^{\mathrm{sys}} \in E^{\mathrm{sys}}_{\varphi'}(\iota, e)$. In this case, take $r(1) = u$.

For the induction step, let $k > 0$, and let $e \in 2^{\mathrm{AP}^{\mathrm{env}}}$. If $e \notin E^{\mathrm{env}}_{\varphi'}(L'(r(k)))$, then any suffix to the play is winning for the system (so construction of r can stop or proceed arbitrarily). Otherwise, there are two cases to consider. First, if $r(k) \in V^i$ for some i, i.e., it is a node that occurs as part of a substrategy A^i used to solve the reachability game $\mathrm{Reach}_{\varphi'}(\mathrm{Entry}^i, \mathrm{Exit}^i)$, then either there is a $u \in V'$ such that

$(r(k), u) \in E(A^i)$ and $L^i(u) \cap \mathrm{AP}^{\mathrm{env}} = e$, or $r(k)$ is a terminal node in A^i and thus can be identified with a node of A that has state label in Exit^i. For the former, by definition of the reachability game $L^i(u) \cap \mathrm{AP}^{\mathrm{sys}} \in E^{\mathrm{sys}}_{\varphi'}(L'(r(k)), e)$, and thus, take $r(k+1) = u$. For the latter, $r(k) \notin U_i$ for any i, hence an appropriate $u \in V$ must exist by hypothesis from the original edgeset $E(A)$. For the second case, i.e., $r(k)$ is not in V^i for any i, $r(k)$ must not be in U_i for any i, so the preceding argument applies, and again we can take $r(k+1) = u$. Therefore, by induction A' has an infinite execution r corresponding to any sequence of inputs that satisfy ρ^{env} at each transition, i.e., δ' is well defined, and A' is a strategy automaton.

It remains to show that RA' is a reach annotation on A' with respect to φ', from which it follows by Theorem 3 that A' is winning for φ'. By hypothesis, the original strategy automaton A has a reach annotation RA and each patch automaton A^i such that $(A^i, \mathrm{Entry}^i, \mathrm{Exit}^i)$ has partial reach annotation RA^i. If describing a strategy automaton, a partial reach annotation may be regarded as a reach annotation with constant RA^i_1. From these observations, it suffices to consider the transitions from A into A^i, and A^i into A. Lines 10–22 of Algorithm 2 ensure that RA_2 is nonincreasing across these transitions. It follows that RA' is a reach annotation on A'. □

3.12 Numerical experiments

Toward practical validation of the proposed algorithms for patching a strategy after changes to reachability in a GR(1) game, in this section experiments are performed using random instances of two motion planning problems on graphs. In both cases, the same basic steps are followed: generate an instance, solve it, modify reachability in that instance, and then attempt to solve it using the methods of this chapter and, separately, using global re-synthesis (no patching).

3.12.1 Gridworlds

An old problem domain in AI and path planning research is gridworlds. In the simplest form, a *gridworld* is a 4-connected undirected graph in which some of the vertices

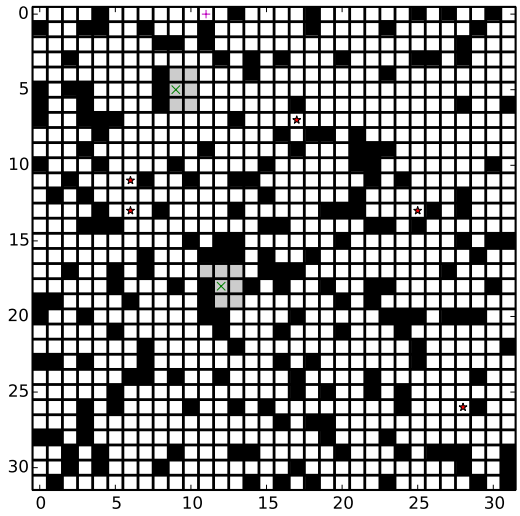

Figure 3.4: Example of a random gridworld with size 32×32 and a block density of 0.2. The initial position is (0,11), which is in matrix-style notation of (row,column). There are 5 randomly placed goal cells that are indicated by small red stars in the plot, and two moving obstacles restricted to the gray regions.

are considered unsafe or blocked. The term 4-connected signifies that all vertices have 4 neighbors, except vertices along the boundary that have 2 or 3 neighbors depending on whether or not it is a corner. The term "grid" emphasizes that the arrangement of vertices is on a uniform grid, e.g., a bounded rectangle in $\mathbb{Z}^2$. For control synthesis subject to temporal logic specifications, the setting of gridworlds is justified by invoking a discrete abstraction to realize physical trajectories among cells; cf. §2.6. As practical motivation, consider that a widely used device for mapping is the occupancy grid, which usually is a uniform grid in which each cell has an associated likelihood of being occupied. Applying a threshold to these likelihoods results in a gridworld because cells likely to be occupied are simply treated as static obstacles, and other regions are regarded as free for motion. Obviously more sophistication can be used, e.g., incorporating likelihoods from the occupancy grid as part of cost for planning motion through the map, but the salient features of gridworlds are still present. Furthermore, slowly moving indoor mobile robots, which usually can rotate in place, easily allow uniform grids for discrete abstractions.

Gridworld generation for each trial is achieved in two parts. First, randomly generate realizable gridworlds of size 32×32 and block density of 0.2, having one initial cell and 5 goal cells, along with 2 moving obstacles. This forms the nominal GR(1) game, i.e., φ_0 in Problem 3. An example of a gridworld thus sampled is shown in Figure 3.4. Each moving obstacle is constrained to motion in a subgrid of size 3×3, and must always eventually return to the center of the subgrid. This assumption of returning to the center prevents arbitrary blocking of the robot and is represented in the GR(1) formula as one of the ψ_j^{env} subformulae. After a nominal realizable gridworld is sampled, a new static obstacle is introduced by blocking a cell that is

- not a static obstacle, i.e., not already an unreachable position;
- not a goal cell, the blocking of which would cause the modified GR(1) game to be unrealizable;
- not the initial cell, the blocking of which would again cause the modified GR(1) game to be unrealizable;
- visited under some play of the strategy.

The final qualifier ensures that the game change actually affects the original strategy automaton in the sense of §3.8. After the location (cell) of the new static obstacle is selected, then the trial proceeds:

1. synthesis for the nominal GR(1) formula is performed;
2. the new static obstacle is introduced;
3. global re-synthesis is performed;
4. patching is performed (separately from the global re-synthesis).

Patching is attempted over subsets of states in progressively larger ∞-norm balls centered at the newly blocked cell and beginning with radius of 1 and continuing to 5 until success. Note that patching may fail for all attempted radii of norm balls. Times required for the different methods are shown in Figures 3.5 and 3.6.

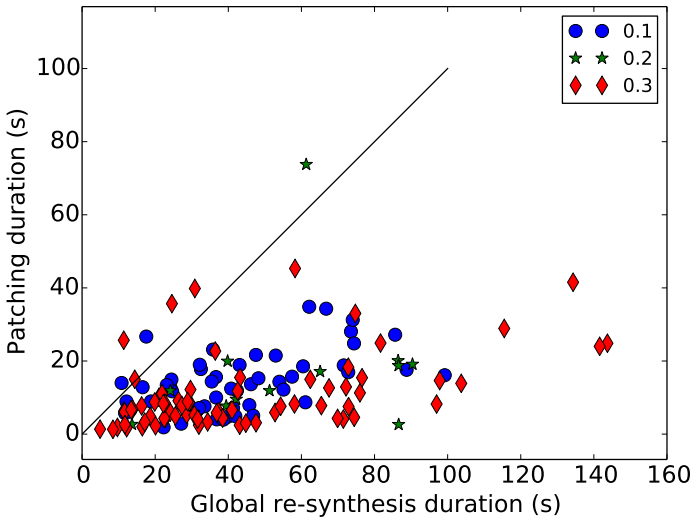

Figure 3.5: Scatter plot of global re-synthesis times against patching times for block densities of 0.1 in 52 trials, 0.2 in 16 trials, and 0.3 in 71 trials. The solid black line has unity slope.

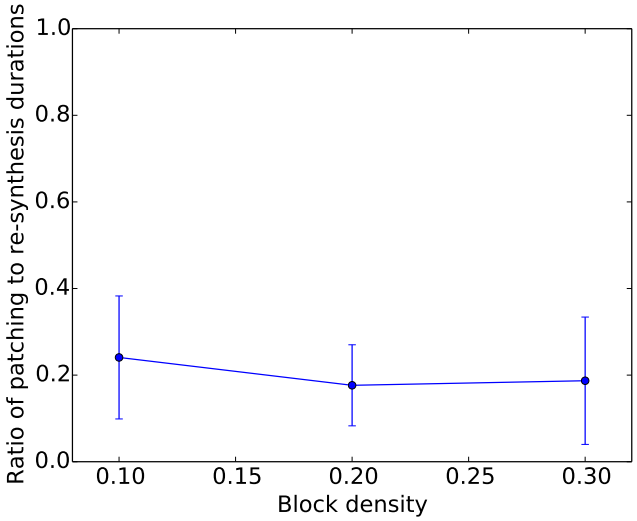

Figure 3.6: Mean values and standard deviations (as error bars) for ratios of patching time to global re-synthesis, plotted with respect to block density. Numbers of trials are 52 trials for density of 0.1, 16 trials for density of 0.2, and 71 trials for density of 0.3.

3.12.2 Random graphs in Euclidean space

The numerical experiments on gridworlds described in the previous section have good practical motivation for robots that have mostly trivial dynamics and are restricted to planar workspaces, e.g., slowly moving outdoor rovers or indoor wheeled service robots. Aerial vehicles and other dynamical systems for which simple kinematic models are not sufficient may require discrete abstractions that are partitions of many polytopes among which having shared facets is not enough to declare existence of transitions. In robotics, workspaces may be discretized using cell decompositions based on the locations and shapes of obstacles, which can be quite different from a uniform, 4-connected grid [37]. The experiment described in this section is an attempt to capture several salient features of discrete abstractions obtained in these challenging settings.

In each trial, an undirected graph is constructed by uniform sampling of n points in the rectangle $[-10, 10]^d \subset \mathbb{R}^d$. Call the resulting set of points V. Edges are created between any two vertices that are within a distance of γ of each other, according to the 2-norm (also known as the Euclidean norm),

$$E_\gamma = \{(u, v) \in V \times V \mid \|u - v\|_2 \leq \gamma\}.$$

A GR(1) game is defined by regarding the vertices as states and the edges as transition rules. In the implementation, vertices are numbered and a single variable represents the current vertex. Edges are expressed as implications, e.g., in the syntax of gr1c (cf. Appendix C), an example of an LTL representation of three outgoing edges is

```
[](v=0 -> (v'=0 | v'=32 | v'=10))
```

which has the intuitive interpretation that, from the vertex 0 (which is a point in $\mathbb{R}^d$), the controller may select actions to transition to vertices 0, 32, or 10 (again, all of which are points in $\mathbb{R}^d$). Next a subset of vertices $g \subset V$ is designated as goals to be visited repeatedly, and a vertex $\iota \in V$ is the initial game state. Finally, the environment is able to transiently declare certain vertices as being unreachable; it

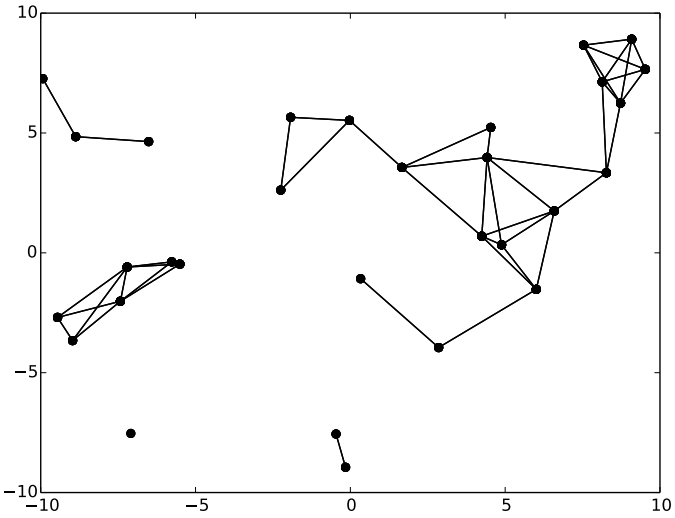

Figure 3.7: Example of a random graph in $\mathbb{R}^2$ with 30 vertices and a maximum connection distance of 4, using the 2-norm. All points are sampled on the rectangle bounded by ± 10 on both axes.

can choose from among the set $M \subset V$. Intuitively, the GR(1) specification thus constructed can be thought of as a game of moving a token on a graph. At each time, the controller chooses the next vertex on which to place the token from among the available outgoing edges. The token must be placed on several of the vertices repeatedly, and certain vertices can become unavailable according to the choice of an adversary.

Examples of random graphs with vertices in $\mathbb{R}^2$ are shown in Figures 3.7 and 3.8. While it is not studied here, it is an interesting direction of future work to explore percolation behavior of these random graphs as recently studied in the context of sampling-based motion planning [29]. Note that gridworlds from the previous section are a special case of these random graphs, except in terms of moves that are available to the adversary. Indeed, let V be the intersection of $\mathbb{Z}^2$ and $[0, c-1] \times [0, r-1]$, and let the maximum connection distance $\gamma = 1$. This yields a 4-connected grid with r "rows" and c "columns." Removing vertices is equivalent to adding static obstacles.

Random trials were performed as described above, varying the number of vertices (n), the number of vertices that the environment can block ($|M|$). To cause a change in reachability in the GR(1) formula, a vertex is randomly selected for removal, provided

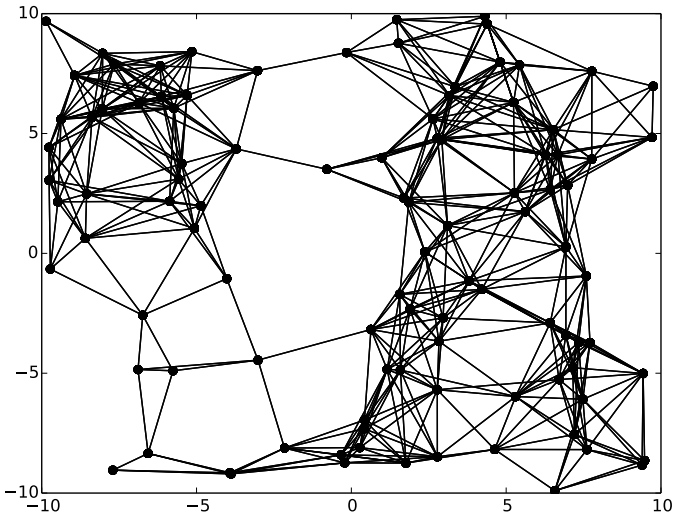

Figure 3.8: Example of a random graph in $\mathbb{R}^2$ with 100 vertices. Compare with Figure 3.7.

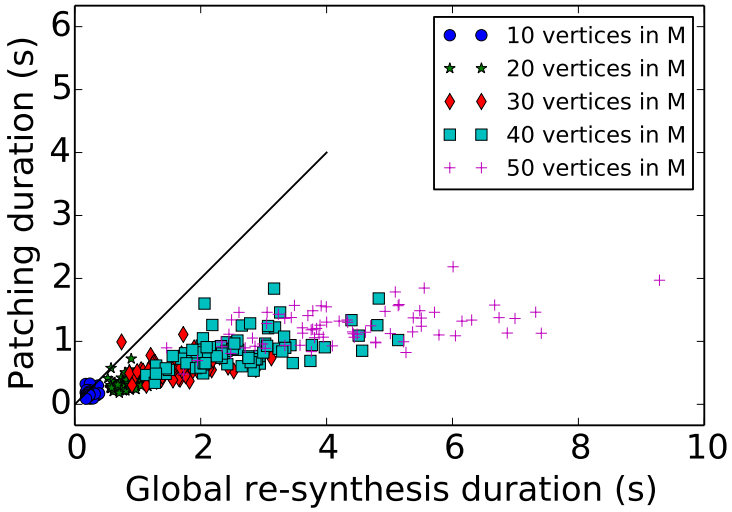

Figure 3.9: Scatter plot of global re-synthesis times against patching times. For every trial, the graph has 200 random vertices. The number that can be adversarially disabled (i.e., the size of M) varies, as indicated in the plot. The maximum distance for edges is $\gamma = 4$. The number of trials per condition 70 trials for $|M| = 10$, 82 trials for $|M| = 20$, 65 trials for $|M| = 30$, 78 trials for $|M| = 40$, and 75 trials for $|M| = 50$. The solid black line has unity slope.

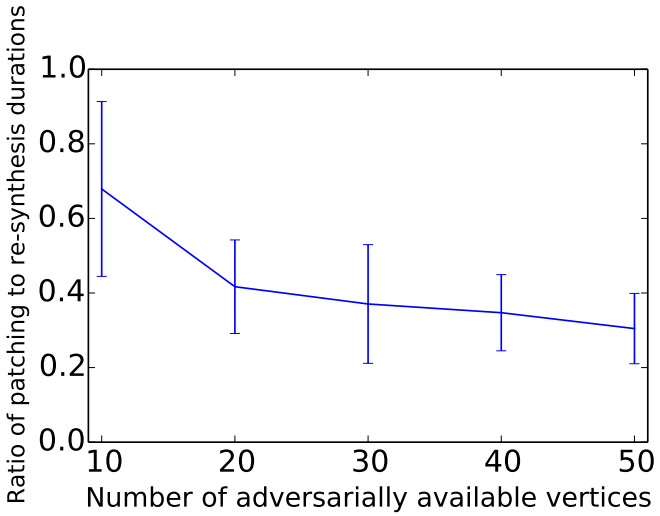

Figure 3.10: Using the same data presented in Figure 3.9, mean ratios of time required to perform patching to that required for global re-synthesis (i.e., discarding the nominal strategy). Error bars indicate 1 standard deviation. The numbers of trials are listed in the caption of Figure 3.9.

that it is not a goal (which would render the modified specification unrealizable), it is not the initial vertex, and that the state (vertex) is visited in some play of the nominal strategy automaton. (This conditions for selecting a state to make unreachable are entirely similar to those of the previous section, §3.12.1.) In all cases, points are in $\mathbb{R}^2$ and the maximum distance for edges (γ) is 4. In each trial, neighborhoods for patching are determined by finding all states (vertices) in a norm ball with center at the newly removed vertex and progressively larger radii, from 0.1 to 10, using 20 steps in total including endpoints. Running times are shown in Figures 3.9 and 3.10.

Algorithm 3 Patching with goal states

INPUT: GR(1) formula φ, strategy A, reach annotation RA, modified formula φ', neighborhood $N \subseteq \Sigma$
OUTPUT: set of tuples $(A^j, \text{Entry}^j, \text{Exit}^j, I_j)$ with partial reach annotation RA^j
`Patches` $:= \emptyset$
for all $i = 0, 1, \ldots, n-1$ //Sort by goal mode **do**
  $N_i := \{u \in N \mid \text{RA}_1(u) = i\}$
end for
$\hat{E}_N := \{(N_i, N_j) \mid N_i \times N_j \cap E \neq \emptyset\}$
$\kappa := 0$
$I_0 := \{0\}$
for all $i = 0, 1, \ldots, n-1$ //Partition indices **do**
  if $(N_i, N_{i+1 \mod n}) \in \hat{E}_N$ **then**
    $I_\kappa := I_\kappa \cup \{i+1 \mod n\}$
    if $i = n-1$ **then**
      $I_0 := I_0 \cup I_\kappa$ //Merge first and last index sets
      $\kappa := \kappa - 1$
    end if
  else
    $\kappa := \kappa + 1$
  end if
end for
for all $j = 0, 1, \ldots, \kappa$ **do**
  $B := \text{Exit}^j$
  set A^j to `nil`
  set i such that $I_j = \{i, i+1, \ldots, i+|I_j|-1\} \mod n$.
  for all `offset` $= |I_j|-2, |I_j|-3, \ldots, 0$ **do**
    $C := [\![\psi^{\text{sys}}_{i+\texttt{offset}}]\!] \cap [\![\text{Local}_{\varphi'}(B)]\!]$
    if $C = \emptyset$ **then**
      abort //Failed to find a substrategy
    end if
    compute strategy $A^j_{C \to B}$ to reach B from C.
    merge $A^j_{C \to B}$ into A^j.
    $B := C$ //Prepare for next loop iteration
  end for
  if $\text{Entry}^j \not\subseteq [\![\text{Local}_{\varphi'}(B)]\!]$ **then**
    abort //Failed to find a substrategy
  end if
  compute $A^j_{\text{Entry}^j \to B}$ to reach B from Entry^j.
  merge $A^j_{\text{Entry}^j \to B}$ into A^j.
  `Patches` $:=$ `Patches` $\cup (A^j, \text{Entry}^j, \text{Exit}^j, I_j)$
end for
return `Patches`

Chapter 4

Patching for Changes in Requirements of Liveness

4.1 Introduction

In the previous chapter, a problem was studied in which a GR(1) game was incrementally changed by modifying transition rules. That is, the safety aspect of the game changed, and thus the set of reachable states that are winning was modified. Algorithms were presented for adjusting (patching) a nominal strategy automaton in response to these changes and thereby obtaining correctness (i.e., a winning strategy) for the sequence of GR(1) formulae. In this chapter, incremental modifications to the other major aspect of the GR(1) formula are addressed, namely the system liveness requirements (or "system goals"):

$$\bigwedge_{i=0}^{n-1} \Box \diamondsuit \psi_i^{\text{sys}}.$$

Remember one of the salient properties of liveness compared to safety: a certificate of violation of liveness cannot be provided as a finite sequence of states (a finite string, using the terminology of LTL from §2.1 and §2.2). Thus, in contrast to the problem setting of the previous chapter, the nominal strategy does not need to be patched immediately. Instead, we can wait and modify the strategy automaton later. This implies a great versatility in applications of the methods for adding and removing

system liveness conditions that are presented in this chapter.

In this chapter, an objective function on discrete states is used as a measure of distance, specifically as part of the algorithm in Section 4.3.2. This could arise, for instance, from a discrete abstraction on the workspace and continuous robot dynamics. Since in the scope of the present work, nothing else is needed from the underlying dynamical system, the construction of a discrete abstraction is not presented (but cf. §2.6 for how one may be constructed). Instead, throughout the chapter reference is made only to "discrete states."

4.2 Problem statements

Before beginning, an important observation can be made about initial conditions and the form of winning strategy automata.

Theorem 13. *Let (ι, φ) be a GR(1) game that is realizable. There exists a winning strategy automaton $A = (V, \{v_0\}, \delta, L)$ such that for every execution r of A, $r(t) \neq v_0$ for $t > 0$.*

Proof. Let $A = (V, \{v_0\}, \delta, L)$ be a winning strategy automaton for the GR(1) game, which must exist by hypothesis (by definition of realizability). Suppose that the desired property does not hold (otherwise A can be used as is), i.e., that for some execution r of A, $r(t) = v_0$ for some $t > 0$. Notice from the definition of executions of strategy automata, $r(0) = v_0$. Define a new strategy automaton $A' = (V \cup \{v_0'\}, \{v_0'\}, \delta', L')$ where for $v \in V \cup \{v_0'\}$

$$L'(v) = \begin{cases} L(v_0) & \text{if } v = v_0' \\ L(v) & \text{otherwise,} \end{cases}$$

and for all $v \in V \cup \{v_0'\}$, $e \in E_\varphi^{\text{env}}(L'(v))$

$$\delta'(v, e) = \begin{cases} \delta(v_0, e) & \text{if } v = v_0' \\ \delta(v, e) & \text{otherwise.} \end{cases}$$

It follows that A' is a winning strategy automaton and that for all executions r of A', $r(t) \neq v'_0$ for $t > 0$. □

By the above theorem, it is without loss of generality that the strategy automaton is taken to have an initial node that is not in a strongly-connected component. In other words, for at least the node from which all executions begin (and possibly others), there is always a finite prefix of game states after which those nodes of the strategy automaton can never be returned to, under any play. Besides being useful in theory, in practical implementations of synthesis algorithms it is always possible to ensure that such a prefix of nodes is present. However, there is also usually a practical motivation to produce small strategies, and one technique to do so is by avoiding a separate initial prefix subgraph of the strategy. Throughout this chapter, we assume such a compression has not been performed. This assumed form of the given strategy automata is crucial for correctness results proven in Sections 4.3.3 and 4.4.3.

Recall the template of GR(1) formulae (2.3) from §2.5,

$$\square\,\rho^{\text{env}} \wedge \left(\bigwedge_{j=0}^{m-1} \square\diamondsuit\,\psi_j^{\text{env}}\right) \implies \square\,\rho^{\text{sys}} \wedge \left(\bigwedge_{i=0}^{n-1} \square\diamondsuit\,\psi_i^{\text{sys}}\right).$$

Throughout the chapter, "goals" or "robot goals" refer to subformulae ψ_i^{sys} appearing on the right-side of the above. As in the previous chapter, specifications for which the only feasible solutions drive the environment to a dead-end are not considered here. Besides not being well-motivated generally, as argued in §2.5, such cases clearly do not involve liveness (being finite strings) and therefore are not relevant for the problems treated in this chapter.

We are now ready to state the problems solved in this chapter. Let (ι, φ) be a GR(1) game, as in (2.3), and let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton that realizes φ.

Problem 4. Given a Boolean formula ζ, defined over the same atomic propositions $\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}$ as φ, find a strategy automaton realizing φ', the modification of φ that includes $\square\diamondsuit\,\zeta$ as a requirement, or determine that φ' is not realizable.

Problem 5. Given an index $i \in \{0, 1, \ldots, n-1\}$, find a strategy automaton realizing φ', the formula obtained by deleting $\square\diamondsuit\psi_i^{\text{sys}}$ from φ, or determine that φ' is not realizable.

4.3 Adding goals

Problem 4 is a precise statement of the intuitive situation where, after having already been given initial safety and liveness requirements about the behavior of a controller, a new set of goal states is declared, and it must be repeatedly visited in addition to original requirements. Practically this could occur because a new region of interest is discovered in a long-running surveillance task. In contrast to Problem 3, only a single modification (addition) occurs to the original GR(1) game. However, the general problem of incremental addition of a sequence of new goals $\zeta_1, \zeta_2, \ldots$ is easily obtained by iterating the statement of Problem 4.

As for the case of a change in reachability, obviously it is possible, upon receiving the new request ζ, to perform global re-synthesis for φ' from scratch, and discard the original strategy automaton without attempting modifications. However, in some cases we can reduce the time and amount of computation required, as demonstrated empirically in §4.5.

4.3.1 Overview

Before presenting the algorithm, the major concepts and steps are described. Let φ be a GR(1) formula with n system goals, i.e., having $\psi_0^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$, and let $A = (V, \{v_0\}, \delta, L)$ be a winning strategy automaton for it. Let ζ be a Boolean (non-temporal) formula in terms of $\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}$, as in the statement of Problem 4. Omitting the transition rules, which are unchanged from φ, the new formula describing assumptions about environment liveness and the requirements of system (robot) goals is

$$\bigwedge_{j=0}^{m-1} \square\diamondsuit\psi_j^{\text{env}} \implies \left(\bigwedge_{i=0}^{n-1} \square\diamondsuit\psi_i^{\text{sys}} \wedge \square\diamondsuit\zeta \right). \tag{4.1}$$

Recall that, by Theorem 4, we can suppose without loss of generality that there is a reach annotation RA on A. (If there were not one for this particular A, we could find a winning strategy automaton that has one.) Informally, the first value RA_1 can be thought of as the index of the goal currently being pursued by the strategy. As such, for a node $v \in V$, $\mathrm{RA}_1(v)$ is referred to as the *goal mode* at v. The motivation for this terminology is that, by Definition 2, RA_1 can only change value when a node where RA_2 is zero is reached, and this corresponds to a i-system goal state. Provided fairness of the environment (i.e., satisfaction of the liveness conditions assumed for it), a state satisfying $\psi^{\mathrm{sys}}_{\mathrm{RA}_1(v)}$ will eventually be reached because A is winning. Upon reaching the goal state, the goal mode is incremented modulo n. Call the set of nodes where a state satisfying ψ^{sys}_i is reached intentionally G_i. The strategy automaton can thus be regarded grossly as moving among sets G_i, for $i \in \{0, 1, \ldots, n-1\}$, in which system (robot) goals from φ are reached, again provided a fair environment. The basic idea of the presented algorithm is to identify which of these sets is most near states that satisfy the new goal ζ and then to pose and solve two reachability games so that ζ-states can be reached and then the original strategy can be returned to. The major steps of the algorithm are as follows.

1. Let Dist() be a function of pairs of discrete states to real numbers. (The name "dist" is suggestive of "distance," but as discussed later, it does not need to be.) Apply Dist() to ζ-states pair with each of original system goal states: ψ^{sys}_0-states, ψ^{sys}_1-states, etc. Choose $i^* \in \{0, \ldots, n-1\}$ achieving the minimum.

2. Find all nodes in the strategy automaton that intentionally reach states satisfying $\psi^{\mathrm{sys}}_{i^*}$ and $\psi^{\mathrm{sys}}_{i^*+1}$ (index addition is modulo n). This "intention" is made precise in the presented algorithm using the reach annotation. Call these sets of nodes G_{i^*} and G_{i^*+1}, respectively. Note that the sets $G_0, \ldots, G_{n-1}$ are not necessarily mutually disjoint. Also, for a winning strategy, it is still possible that some $G_i = \emptyset$, because a winning strategy could involve one of the assumed environment liveness conditions eventually being persistently not satisfied.

3. Solve two reachability games in sequence, first from G_{i^*} (or rather, states with

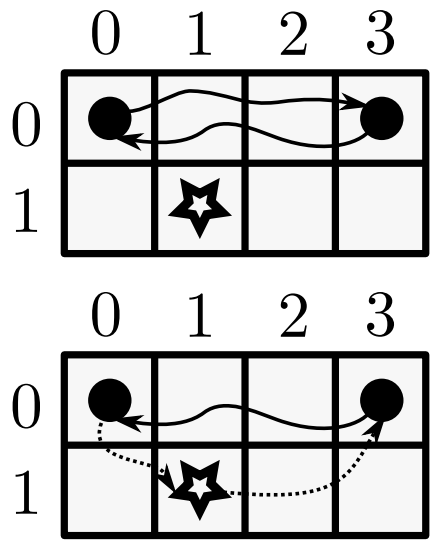

Figure 4.1: Illustrative deterministic (without adversarial environment) example for Algorithm 4. In both panels, a 2×4 grid is shown. The original task is to visit cells $(0, 0)$ and $(0, 3)$ repeatedly. In both panels, solid black circles are shown to indicate locations where there is a node of the strategy automaton reaching a desired state. In the top panel, the nominal strategy is indicated with solid curves between the nodes to indicate possible trajectories by the underlying dynamical system, e.g., a double-integrator. The star in cell $(1, 1)$ is introduced as a new region to be repeatedly visited. In the bottom panel, one of the nominal trajectories is deleted and new (dotted) curves illustrate the patch.

which strategy nodes in G_{i^*} are labeled) to ζ-states, and then from those ζ-states that were reached to G_{i^*+1}.

4. Delete existing nodes and edges in A that lead to G_{i^*+1} from G_{i^*}. Append strategies for the two reachability games of the previous step, thereby obtaining the patched strategy automaton A'.

A small example intended to illustrate the algorithm for handling new goals (Problem 4) is shown in Figure 4.1. The example is deterministic because there is no adversary; motion by the robot is simply based on adjacency, as in a 4-connected grid. Recall that the availability of a discrete abstraction (cf. §2.6) is assumed, and thus motion among cells is concretely realized using some other controllers that are known to exist as part of the abstraction. The curves in the figure illustrate physical trajectories. The original task, as a GR(1) game, is to visit cells $(0, 0)$ and $(0, 3)$

repeatedly. (Coordinates follow the common matrix convention of row, column.) Let v_0 be the node of the strategy automaton corresponding to occupancy of cell $(0, 0)$, and let v_1 be the node corresponding to cell $(0, 3)$. Notice that there may be multiple nodes labeled with cells $(0, 1)$ and $(0, 2)$ because the action to take from those states depends on history, or rather, the current goal mode. For this example, the algorithm proceeds as follows.

1. Using the Euclidean distance between centers of cells for Dist(), the goal cell $(0, 0)$ is closer to the new goal $(1, 1)$ than the other original goal cell $(0, 3)$. As such, take $i^* = 0$.

2. Clearly $G_0 = \{v_0\}$ and $G_1 = \{v_1\}$.

3. The reachability game to go from $L(G_0) = \{(0, 0)\}$ to the set of ζ-states, i.e., $\{(1, 1)\}$, is solved by a simple finite strategy moving down and to the right.

4. A similar strategy drives the robot from $(1, 1)$ to $(0, 3)$, thereby solving the second reachability game to $L(G_1)$.

5. The solutions for the two reachability games are merged into the original strategy automaton, and nodes that would drive to $(0, 3)$ from $(0, 0)$ are safely deleted.

4.3.2 Algorithm

The method providing a solution for Problem 4 is presented in Algorithm 4. Explanations of several details are as follows.

- Line 3: Despite the suggestive name, Dist() does not need to be a metric or otherwise provide some formal notion of distance. Instead, it can be thought of as an objective function to be used when deciding where to insert solutions for the reachability games in which ζ-states are visited. Correctness of the algorithm is not affected by this, and thus when no meaningful Dist is known, it can simply be a constant (then the choice of i^* is arbitrary).

Algorithm 4 Append a new goal ζ

1: INPUT: strategy automaton $A = (V, I, \delta, L)$, reach annotation RA, distance function Dist, Boolean formula ζ
2: OUTPUT: augmented automaton A' and reach annotation RA$'$
3: $i^* := \operatorname{argmin}_{i=0,1,\ldots,n-1} \mathrm{Dist}\,(\psi_i^{\mathrm{sys}}, \zeta)$.
4: $G_{i^*} := \emptyset$
5: **for all** $v \in V$ **do**
6: **for all** $u \in \mathrm{Pre}(v)$ **do**
7: **if** $\big(\mathrm{RA}_1(u) < \mathrm{RA}_1(v) \wedge \mathrm{RA}_1(u) \leq i^* \wedge \mathrm{RA}_1(v) > i^*\big)$ $\vee\big(\mathrm{RA}_1(u) > \mathrm{RA}_1(v) \wedge (\mathrm{RA}_1(u) \leq i^* \vee \mathrm{RA}_1(v) > i^*)\big)$ **then**
8: **if** $\mathrm{RA}_2(u) = 0$ **then**
9: $G_{i^*} := G_{i^*} \cup \{u\}$
10: **else**
11: $G_{i^*} := G_{i^*} \cup \{v\}$
12: **end if**
13: **break** //Skip to next iteration of outer for-loop
14: **end if**
15: **end for**
16: **end for**
17: construct the set G_{i^*+1} in an entirely similar manner to G_{i^*}, but for mode $i^* + 1$.
18: **if** $\mathrm{Reach}_{\varphi'}(G_{i^*}, [\![\zeta]\!])$ is realizable **then**
19: synthesize strategy automaton $A_{i^* \to \zeta}$ for reachability game $\mathrm{Reach}_{\varphi'}(G_{i^*}, [\![\zeta]\!])$.
20: **else**
21: **abort**
22: **end if**
23: set G_ζ to all nodes in $A_{i^* \to \zeta}$ that do not have outgoing edges.
24: **if** $\mathrm{Reach}_{\varphi'}(G_\zeta, G_{i^*+1})$ is realizable **then**
25: synthesize strategy automaton $A_{\zeta \to i^*+1}$ for $\mathrm{Reach}_{\varphi'}(G_\zeta, G_{i^*+1})$.
26: **else**
27: **abort**
28: **end if**
29: $V := V \setminus \mathrm{RA}_1^{-1}(i^* + 1)$
30: $V' := V \cup V_{i^* \to \zeta} \cup V_{\zeta \to i^*+1}$
31: set L' and δ' consistent with appending $A_{i^* \to \zeta}$ and $A_{\zeta \to i^*+1}$ to A.
32: define RA$'$ on V' so that it agrees with RA on V, $\mathrm{RA}_{i^* \to \zeta}$ on $V_{i^* \to \zeta}$, and $\mathrm{RA}_{\zeta \to i^*+1}$ on $V_{\zeta \to i^*+1}$.

- Lines 4–16: Informally, G_{i^*} is the set of nodes that satisfy the system goal $\psi_{i^*}^{\mathrm{sys}}$ and where that was the intent of the strategy automaton. The motivation for using the word "intent" is the pattern of values in the reach annotation that led to the $\psi_{i^*}^{\mathrm{sys}}$-states, which is expressed by the complicated conditional statement.

Consult Definition 2 from §3.6 for details.

- Lines 21, 27: If one of the reachability games does not have a solution, then abort. Note that patching here may fail despite the modified GR(1) game being realizable. Discussion about completeness is given in §4.3.3.

- Line 29: Delete nodes from the original strategy that are no longer used because solutions from the reachability games are used instead. Note that deleting old paths in A between nodes reaching goal $\psi_{i^*}^{\text{sys}}$-states and nodes reaching $\psi_{i^*+1}^{\text{sys}}$-states is easily achieved since it suffices to delete original nodes annotated with mode i^* (and we are given RA, which provides this information).

- Lines 29–32: The final steps produce a new strategy automaton A' and a reach annotation RA' on it by patching to the original the substrategies $A_{i^*\to\zeta} = (V_{i^*\to\zeta}, I_{i^*\to\zeta}, \delta_{i^*\to\zeta}, L_{i^*\to\zeta})$ and $A_{\zeta\to i^*+1}$ found in the algorithm when solving the reachability games. The new labeling $L' : V' \to \Sigma$ is built directly from the components,

$$L'(v) = \begin{cases} L(v) & \text{if } v \in V, \\ L_{i^*\to\zeta}(v) & \text{if } v \in V_{i^*\to\zeta}, \\ L_{\zeta\to i^*+1}(v) & \text{otherwise,} \end{cases} \tag{4.2}$$

for $v \in V'$. Notice that the node sets for the component strategies are disjoint from each other and that of the original strategy automaton, i.e., V' is a disjoint union of V, $V_{i^*\to\zeta}$, and $V_{\zeta\to i^*+1}$. RA' is defined similarly:

$$\text{RA}'(v) = \begin{cases} \text{RA}(v) & \text{if } v \in V, \\ \text{RA}_{i^*\to\zeta}(v) & \text{if } v \in V_{i^*\to\zeta}, \\ \text{RA}_{\zeta\to i^*+1}(v) & \text{otherwise.} \end{cases} \tag{4.3}$$

Details for the creation of δ' are entirely similar to those given in the previous chapter, in particular Algorithm 2, and thus are omitted here.

4.3.3 Results

Correctness of Algorithm 4 is proven in this section. As part of the theorem, it is also shown that the output provides a reach annotation, which is of separate interest because it follows that the same algorithm or the others in this chapter or the previous chapter can be used again, iteratively as reachability or liveness requirements change. The section concludes about completeness.

Theorem 14. *Let $A = (V, \{v_0\}, \delta, L)$ be a strategy automaton that is winning for a GR(1) game (ι, φ), and which has a reach annotation* RA*. Let ζ be a Boolean formula over the same variables as φ, and let φ' be the modification of φ to include $\square\diamondsuit\zeta$ as a requirement. If Algorithm 4 returns a strategy automaton $A' = (V', \{v_0\}, \delta', L')$ and a map* RA$'$*, then they are correct with respect to φ', i.e., A' realizes φ' and* RA$'$ *is a reach annotation on A' for φ'.*

Proof. The proof proceeds in two steps, first showing that all plays under A' are infinite, and then showing that RA$'$ is a reach annotation, from which correctness follows by Theorem 3 from §3.6. First, observe that φ' has the same initial conditions and transition rules as φ. We prove by induction on the graph structure of A' that all plays are infinite. For any initial state of φ' (which corresponds exactly to initial states of φ) we can select an initial node v_0 such that $L(v_0)$ is equal to this initial state. Furthermore, by assumption of the form of A described in §4.2, this same node is preserved during the construction of A' from A because it occurs in a prefix of A (i.e., can occur at most once in any play). Since the transition rules are unchanged in φ', the hypothesis that A realizes φ implies that, for any environment move from the state $L(v_0)$, there is an outgoing edge (i.e., a robot move) from v_0 consistent with the transition rules of φ'; otherwise, there would be a play in A that is finite, contradicting the hypothesis of its correctness with respect to φ. For the induction step, let v_k be a node in A' reached under a play $\sigma_0 \cdots \sigma_k$ (so that $L(v_k) = \sigma_k$) that is consistent with the initial conditions and transition rules of φ' (which are the same as those of φ). From Algorithm 4, this node is either a node originating from A, or from one of the substrategies $A_{i^* \to \zeta}$ or $A_{\zeta \to i^*+1}$. In the first case, by hypothesis of A

realizing φ, for every possible environment move from σ_k, there is an outgoing edge from v_k consistent with the transition rules of φ', leading to a node v_{k+1} in A'. In the latter case, the definition of reachability games $\text{Reach}_{\varphi'}$ (cf. §3.7) ensures that every possible move by the environment under φ' from the state $L(v_k)$, and thus also under φ, has a corresponding outgoing edge in $A_{i^* \to \zeta}$ from node v_k. Mutatis mutandis for $A_{\zeta \to i^*+1}$. Therefore by induction we conclude that all plays of A' are infinite.

Next we show that RA' is a reach annotation for A' with respect to task formula φ'. Let i^* be the index of the goal immediately following which the substrategy reaching ζ-states, $A_{i^* \to \zeta}$, is used. While not done explicitly in Algorithm 4, to fully conform with the definition of reach annotation (cf. Definition 2 from §3.6) we adjust all goal indices (modes) as follows. Define $\psi_n^{\text{sys}} := \zeta$, and let ξ be the permutation of $\{0, 1, \ldots, n-1, n\}$ defined by

$$\xi(\hat{i}) = \begin{cases} \hat{i} & \text{if } \hat{i} \leq i^* \\ n & \text{if } \hat{i} = i^* + 1 \\ \hat{i} - 1 & \text{otherwise.} \end{cases} \tag{4.4}$$

The robot goals can now be expressed in the usual form, as part of φ,

$$\bigwedge_{\hat{i}=0}^{n} \square \diamondsuit \psi_{\xi(\hat{i})}^{\text{sys}}. \tag{4.5}$$

Then RA' is a reach annotation using the modes according to (4.5), i.e., as provided by the permutation (4.4). Explicitly, define the function RA'' to coincide with RA' on the second part, i.e.,

$$\text{RA}''_2(v) = \text{RA}'_2(v)$$

for all $v \in V'$, and to use permuted modes from RA' on the first part, i.e.,

$$\text{RA}''_1(v) = \xi^{-1}(\text{RA}'_1(v)). \tag{4.6}$$

Finally, Algorithm 4 appends substrategies for entire modes, i.e., all original nodes

with mode $i^* + 1$ are deleted, and the substrategies $A_{i^* \to \zeta}$ are $A_{\zeta \to i^*+1}$ are always followed in sequence, and their nodes have modes n and i^*+1, respectively. Therefore, RA'' is a reach annotation for A' with respect to φ'. Since RA' is the same as RA'' up to a permutation of goal modes, we have that RA' is also a reach annotation for A'. From Theorem 3, it follows that A' must be winning, hence it must realize φ', concluding the proof. □

The addition of a distinct system goal, i.e., where $\zeta \not\equiv \psi_i^{\mathrm{sys}}$ for all i, results in a GR(1) game that is harder than the original in that a winning strategy for the new game is necessarily winning for the original, but the converse does not hold. Intuitively this is true because visitation of ζ-states can simply be ignored when reasoning about correctness for the original game. In summary, we have

Remark 15. Any play that is correct with respect to φ' is correct with respect to φ.

In terms of formal language inclusion, $\mathcal{L}(\varphi') \subseteq \mathcal{L}(\varphi)$.

Theorem 16. *Algorithm 4 is not complete.*

Proof. Consider the directed graph $\mathcal{G}$ having set of vertices $\{\mathrm{s1}, \mathrm{s2}, \mathrm{s3}, \mathrm{s4}, \mathrm{s5}\}$. The set of edges is that which minimally satisfies the following. $(\mathrm{s1}, \mathrm{s2})$ and $(\mathrm{s1}, \mathrm{s3})$ are edges. The subgraphs induced by $\{\mathrm{s2}, \mathrm{s4}\}$ and $\{\mathrm{s3}, \mathrm{s5}\}$ are complete. (Cf. Figure 4.2). Let the vertices of $\mathcal{G}$ be identified with atomic propositions, and consider the formula ρ^{sys} defined such that precisely one of $\mathrm{s1}, \ldots, \mathrm{s5}$ is `true` at each time. Take as initial state $\{\mathrm{s1}\}$. Now, defining $\psi_0^{\mathrm{sys}} = s3 \vee s4$, a GR(1) game is obtained that lacks adversarial environment and has one system liveness requirement (robot goal). Clearly a winning strategy is to move from s1 to s3 and then remain in the subgraph induced by $\{\mathrm{s3}, \mathrm{s5}\}$. Consider a new liveness requirement $\zeta = s2$. Patching of the original strategy is not possible. Since the strategy is fixed before ζ is known, this situation cannot be avoided. □

It should be apparent that the counter-example is not special to Algorithm 4 and could serve to demonstrate that other methods for incrementally adding liveness

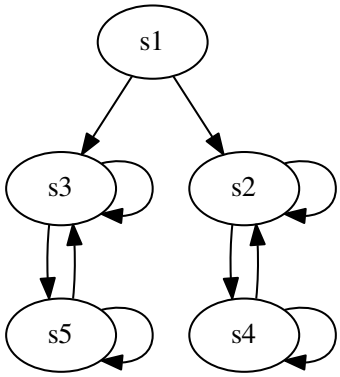

Figure 4.2: Depiction of the graph used in the proof of Theorem 16.

properties are not complete. However, the extent to which this is a practical limitation in mobile robotics is not apparent (the counter-example seems contrived).

4.4 Removing goals

The inverse of goal addition is goal removal, and accordingly Problem 5 can be thought of as a complement to Problem 4. However, removal of a system liveness requirement (goal) results in a task that is easier because, as made precise later (§4.4.3, the same strategy can continue to be used, unaltered in the new game. Nonetheless, for long-running or indefinite tasks that involve many instances of Problems 4 and 5, not pruning unnecessary parts of a strategy would allow it to grow without bound. Moreover, the computational complexity of patching algorithms in this chapter and the previous scale with the size of the strategy automata—another reason to prune if possible. Algorithm 5 solves Problem 5 by deleting unnecessary parts of the strategy.

4.4.1 Overview

Before presenting the algorithm, a conceptual overview is given. Let φ be a GR(1) formula with n system goals, i.e., having $\psi_0^{\text{sys}}, \ldots, \psi_{n-1}^{\text{sys}}$, and let $A = (V, \{v_0\}, \delta, L)$

Algorithm 5 Remove an existing goal ψ_i^{sys}

1: INPUT: strategy automaton $A = (V, I, \delta, L)$, reach annotation RA, goal index i
2: OUTPUT: pruned automaton A' and reach annotation RA$'$
3: construct the sets G_{i-1}, G_{i+1} in an entirely similar manner as in lines 4–16 of Algorithm 4.
4: synthesize reach strategy automaton $A_{i-1\to i+1}$ for the reachability game $\text{Reach}_{\varphi'}(G_{i-1}, G_{i+1})$.
5: $V := V \setminus (\text{RA}_1^{-1}(i) \cup \text{RA}_1^{-1}(i+1))$
6: $V' := V \cup V_{i-1\to i+1}$
7: define L', δ' consistent with previous line.
8: define RA$'$ to agree with RA on V and $\text{RA}_{i-1\to i+1}$ on $V_{i-1\to i+1}$.

be a winning strategy automaton for it. Let i be the index of the goal to remove, as in the statement of Problem 5. Omitting the transition rules, which are unchanged from φ, the new formula describing assumptions about environment liveness and the requirements of system (robot) goals is

$$\square\diamondsuit\psi_0^{\text{sys}} \wedge \cdots \wedge \square\diamondsuit\psi_{i-1}^{\text{sys}} \wedge \square\diamondsuit\psi_{i+1}^{\text{sys}} \wedge \cdots \wedge \square\diamondsuit\psi_{n-1}^{\text{sys}}. \tag{4.7}$$

Recall from the discussion in Section 4.3.1 that without loss of generality there is a reach annotation on A. Furthermore, the first part of it, RA_1, indicates the goal mode for each node, which intuitively is the index of the liveness requirement currently being sought. Thus, in order to remove any behavior in pursuit of ψ_i^{sys} by the given strategy, it is enough to delete those nodes have RA_1-value of i and then to patch the strategy with the solution of a reachability game between goals $i-1$ and $i+1$ (modulo n).

4.4.2 Algorithm

The presented solution for Problem 5 is Algorithm 5. Several details are as follows.

- Line 4: This reachability game always has a winning system strategy. Explanation is given in §4.4.3.
- Line 5–8: The process of patching with the strategies from reachability games is described in the previous chapter, in particular Algorithm 2.

4.4.3 Results

In this section, correctness and completeness are presented for Algorithm 5, and the output is shown to provide a reach annotation. As was the case for the results about Algorithm 4 for adding goals (cf. §4.3.3), producing a reach annotation is useful to not only verify that the modified strategy automaton is winning but also so that the other methods in this and the previous chapters can be applied, incrementally correcting a strategy as the game changes in various ways.

In contrast to the first problem, unnecessary nodes in a strategy automaton do not need to be pruned after a liveness requirement is removed. Compare the following with Remark 15.

Remark 17. Any play that is correct with respect to φ is correct with respect to φ'.

Lemma 18. *The reachability game* $\mathrm{Reach}_{\varphi'}(G_{i-1}, G_{i+1})$*, appearing on line 4 of Algorithm 5 is always realizable, i.e., there is at least one substrategy solving it under the transition rules of* φ'

Proof. Recall that the removal of a robot goal from the task does not alter transition rules, nor initial conditions, nor environment liveness assumptions. Therefore, if a state s can be driven by some (finite) strategy to a state t under formula φ, then it can also be done under formula φ' using the same strategy. By hypothesis, the original automaton A realizes φ and in particular reaches G_{i+1} from G_{i-1}, or else blocks an environment liveness condition ψ_j^{env}, and therefore A itself provides a feasible solution to $\mathrm{Reach}_{\varphi'}(G_{i-1}, G_{i+1})$. □

Theorem 19. *Let* A *be a strategy automaton realizing* φ *with reach annotation* RA*. Let* $i \in \{0, 1, \ldots, n-1\}$ *be the index of the goal* ψ_i^{sys} *to be deleted from* φ*, yielding the new task* φ'*. Then* A' *returned by Algorithm 5 on these inputs realizes* φ'*, and* RA' *is a reach annotation.*

Given its similarity to Theorem 14, only a sketch of a proof is given. First, observe that all plays are infinite, i.e., there are no dead-ends in the automaton A. This is true because the original automaton A has infinite plays from hypothesis, only nodes

labeled (via RA) for the deleted goal mode and its successor are removed from A, and the nodes that temporarily have no outgoing edges are connected by a solution of a reachability game that is guaranteed by Lemma 18 to exist. Since all plays are infinite, the second step is to verify that RA' is indeed a reach annotation. This can be shown in a similar manner to that used in the proof of Theorem 14, but now instead of a permutation of modes, we use an injection to account for the gap in the sequence of goal modes due to deletion.

Theorem 20. *Algorithm 5 is complete, i.e., if φ' is realizable, then Algorithm 5 will find a strategy automaton A' realizing it.*

Proof. It is enough to ensure that Algorithm 5 will terminate with *some* automaton A', because then, by Theorem 19 A' must be correct. This is immediate because Algorithm 5 contains nothing more than for-loops and a reachability game a solution for which always exists by Lemma 18. □

Though using Algorithm 5 for pruning is practically motivated by maintaining succinct strategies, there is no guarantee of optimality. Indeed, a cost function was not introduced in this chapter. However, the existing strategy A, which is valid for the modified GR(1) formula φ' by Remark 17, provides an upper bound on size. Nonetheless, intuitively Algorithm 5 should produce strategies that are smaller when there are shorter paths from states satisfying $\psi_{i-1}^{\mathrm{sys}}$ to $\psi_{i+1}^{\mathrm{sys}}$. Indeed, the usual implementation of GR(1) synthesis finds a path to goal states in a typical repeated graph predecessor set computation, as outlined in §3.5.

4.5 Numerical experiments

Toward practical validation of the proposed algorithms for patching a strategy after changes to liveness requirements in a GR(1) game, in this section two experiments are performed. These are exactly the two settings used for experiments of the previous chapter in §3.12, but with a few modifications for the present problems of goal addition and removal.

4.5.1 Gridworlds

Similar to the experiment described in §3.12.1, random gridworlds are generated and GR(1) formulae are created to express requirements of visiting certain cells repeatedly while avoiding collisions with adversarial moving obstacles. Recall the example given in Figure 3.4.

For all trials, the gridworlds have size 32×32. There is one initial cell, from which the robot begins all plays, and there is 1 or 2 moving obstacles that are constrained to a subgrid of size 3×3 that is randomly placed. The block densities, i.e., the proportion of cells that are randomly chosen to be static obstacles, are 0.2 and 0.3. Each trial is performed as follows. Gridworlds are randomly generated until a realizable one is found with the previously described parameters and 10 randomly placed goals cells that are to be visited infinitely often. Then, 9 of the goals are temporarily removed to create a problem instance having only one goal cell (i.e., one position in the gridworld to be repeatedly visited). This is solved to create the original GR(1) formula φ of Problem 4 and a winning strategy automaton together with reach annotation on it. Then, a sequence of applications of Algorithm 4 and global re-synthesis are performed as each of the 9 goals is added back to the problem, where each addition in turn is ζ of Problem 4.

Mean times with standard deviation bars are shown in plots of Figures 4.3, 4.4, 4.5, showing cases of 1 or 2 moving obstacles and block densities of 0.2 and 0.3. Means of ratios of patching to global re-synthesis times are plotted in Figure 4.6, again with bars showing standard deviation.

4.5.2 Random graphs in Euclidean space

Motivated to address some limitations of gridworlds as models of practical applications, in this section an experiment is described in which random graphs are created in $\mathbb{R}^2$. The definitions are the same as those of the previous chapter in §3.12.2. However, the game created in each trial is analogously created to those of the previous section: the randomly created realizable instance has g goals; all but one of these

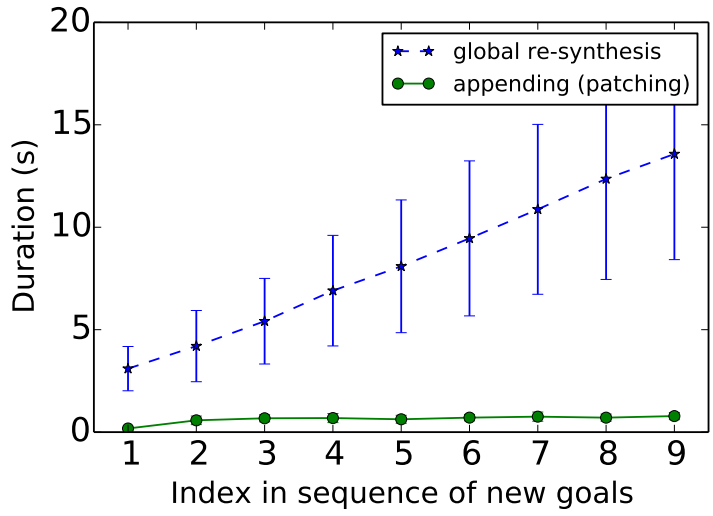

Figure 4.3: Run-times (durations) for applying Algorithm 4 and global re-synthesis (i.e., discarding the original strategy) for the case of 1 moving obstacle and block density of 0.2. Data are from 10 trials. Points are mean values, and error bars are 1 standard deviation.

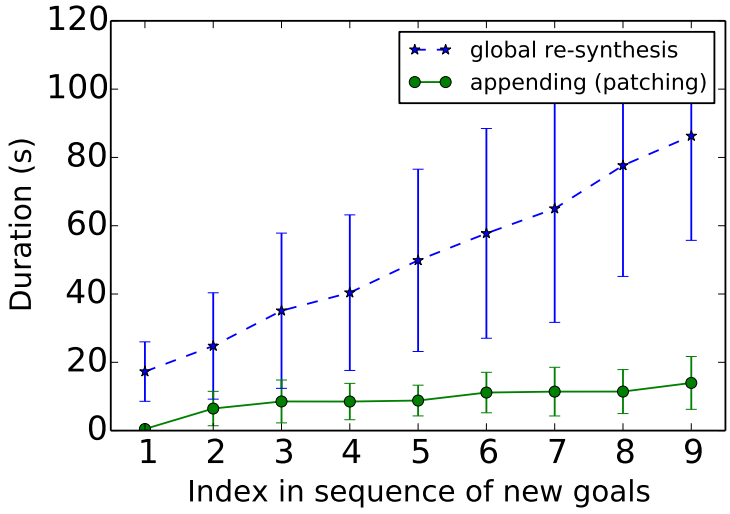

Figure 4.4: Run-times (durations) for applying Algorithm 4 and global re-synthesis (i.e., discarding the original strategy) for the case of 2 moving obstacles and block density of 0.2. Data are from 12 trials. Points are mean values, and error bars are 1 standard deviation.

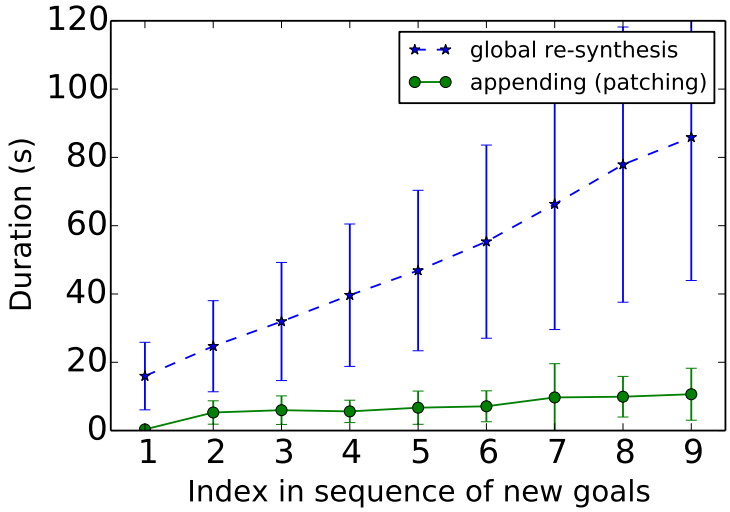

Figure 4.5: Run-times (durations) for applying Algorithm 4 and global re-synthesis (i.e., discarding the original strategy) for the case of 2 moving obstacles and block density of 0.3. Data are from 21 trials. Points are mean values, and error bars are 1 standard deviation.

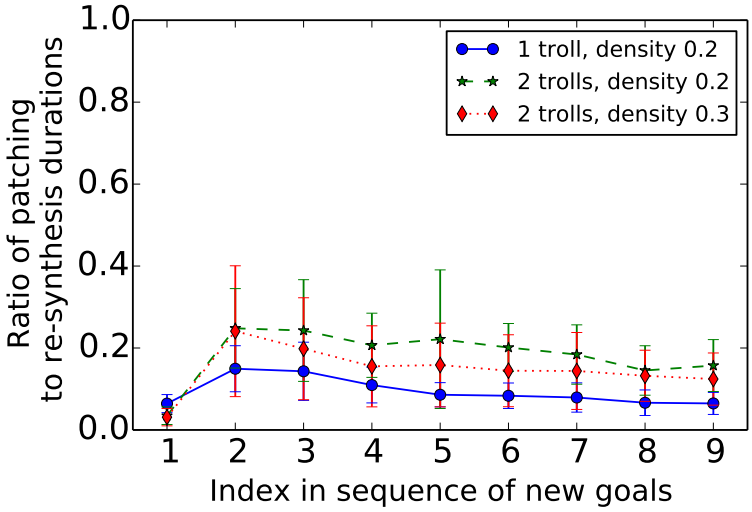

Figure 4.6: Ratios of run-times for applying Algorithm 4 vs. global re-synthesis. Data for each case are as presented separately in Figures 4.3, 4.4, 4.5; in summary, there are 10 trials for 1 moving obstacle ("troll"), block density 0.2; 12 trials for 2 moving obstacles, block density 0.2; 21 trials for 2 moving obstacles, block density 0.3.

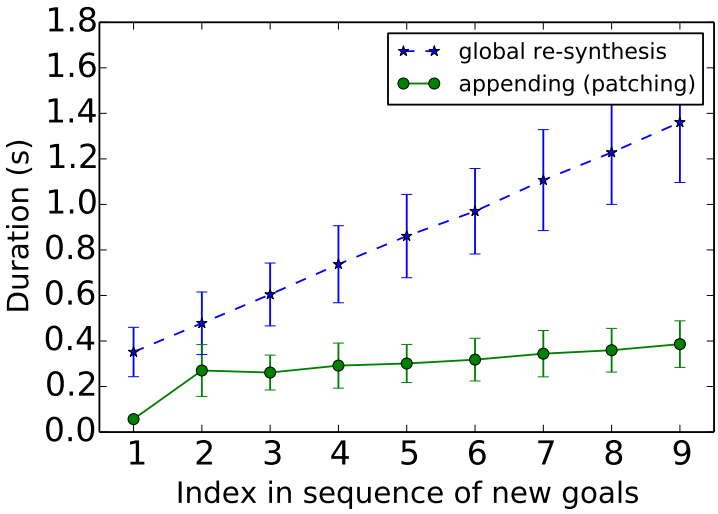

Figure 4.7: Run-times (durations) for applying Algorithm 4 and global re-synthesis (i.e., discarding the original strategy) for the case of 20 vertices that can be adversarially disabled (i.e., the size of M), out of the graph of 200 random vertices in $\mathbb{R}^d$. The maximum distance for edges is $\gamma = 4$. Data are from 150 trials.

are removed to obtain the original problem, and then each is added back in turn, resulting in a sequence of goal additions (each being an instance of Problem 4).

Mean times with standard deviation bars are shown in plots of Figures 4.7 and 4.8, and ratios of times are shown in Figure 4.9.

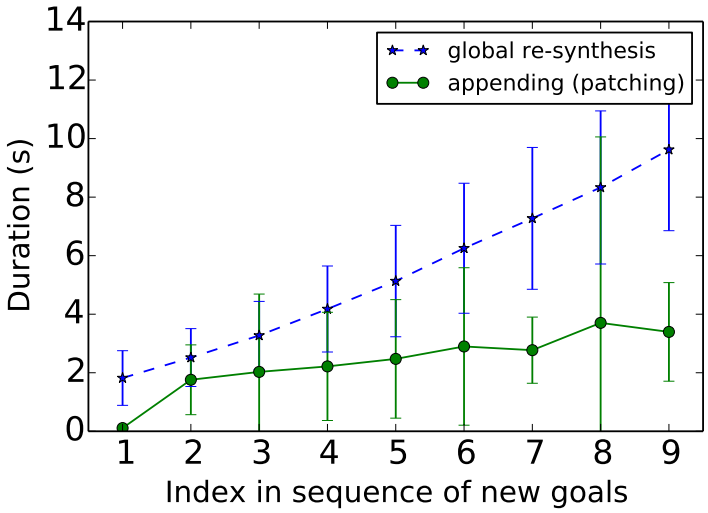

Figure 4.8: Run-times (durations) for applying Algorithm 4 and global re-synthesis (i.e., discarding the original strategy) for the case of 50 vertices that can be adversarially disabled (i.e., the size of M), out of the graph of 200 random vertices in $\mathbb{R}^d$. The maximum distance for edges is $\gamma = 4$. Data are from 150 trials.

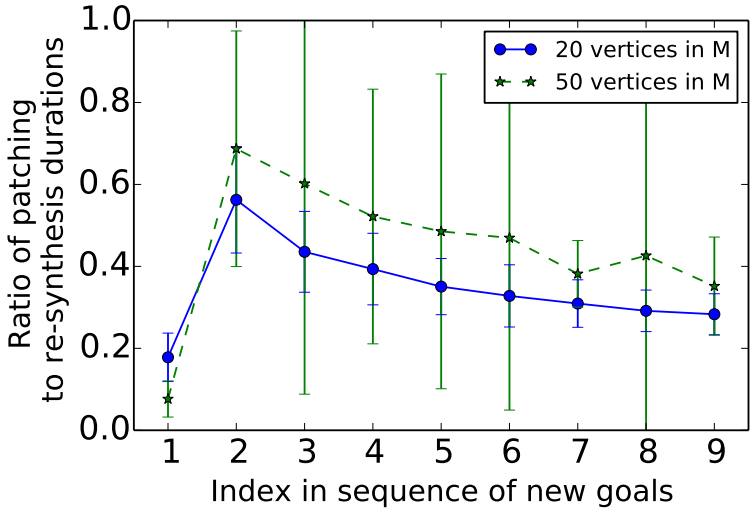

Figure 4.9: Ratios of run-times for applying Algorithm 4 vs. global re-synthesis. Data for each case are as presented separately in Figures 4.7 and 4.8.

Chapter 5

Cross-entropy Motion Planning for LTL Specifications

5.1 Introduction

In the previous two chapters, the problems and solutions have been entirely in terms of systems that can be described by finitely many states. Practically this requires a discrete abstraction, or rather, bisimulation in order to apply the synthesized controller to the actual dynamical system. While there are methods for constructing abstractions, e.g., as in the case of piecewise linear systems as shown in §2.6, in some situations this is not tractable or not possible. Motivated by nonlinear dynamics and labelings of state spaces that cannot be expressed as (nor usefully approximated as) unions of polytopes, in this chapter control is achieved using stochastic optimization for sampling-based trajectory generation.

The methods presented in this chapter build on the cross-entropy (CE) method [17]. The basic CE-LTL algorithm that is presented first is essentially a generalization of the CE motion planning methods introduced by Kobilarov [33]. Parts of this chapter are based on joint work with Wolff [44].

5.2 Control system model and problem formulation

5.2.1 Dynamics and labeling of states

Consider the discrete-time control system defined by the difference equation

$$x(t+1) = f(x(t), u(t)), \tag{5.1}$$

where $x(t) \in X$ are states, $u(t) \in U$ are control inputs, and $t \in \mathbb{N}$. A state $x_0 \in X$ is designated as the *initial state*. Let AP be a set of atomic propositions. Together with the dynamics, a labeling $L : X \to 2^{\mathrm{AP}}$ is given that assigns a set of atomic propositions to each state, thus indicating which are `true` at that state (as in previous chapters). A *trajectory* of system (5.1) is a pair $(\mathbf{x}, \mathbf{u})$ of sequences of states and inputs, $\mathbf{x} : \mathbb{N} \to X$ and $\mathbf{u} : \mathbb{N} \to U$, such that for $t \geq 0$, $\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{u}(t))$ and $\mathbf{x}(0) = x_0$. The *trace* of a trajectory is the sequence of state labels

$$\begin{aligned} \mathcal{L}((\mathbf{x}, \mathbf{u})) : \mathbb{N} &\to 2^{\mathrm{AP}} \\ t &\mapsto L(\mathbf{x}(t)). \end{aligned}$$

Notice that, while $\mathbf{u}$ is considered to be a part of the trajectory, control inputs are not labeled and thus do not directly affect the trace. The treatment given here can easily be extended to the case of labeled inputs.

5.2.2 Problem statement

Let J be a cost function that maps trajectories (state and control input sequences) to nonnegative reals.

Problem 6. Let φ be an LTL formula, and let $x_0 \in X$ be the initial state. Find a trajectory $(\mathbf{x}, \mathbf{u})$ that minimizes $J(\mathbf{x}, \mathbf{u})$ subject to $\mathcal{L}((\mathbf{x}, \mathbf{u})) \models \varphi$.

Though concise and general, this problem statement presents several difficulties.

No assumptions have been made on the difference equation (5.1) nor on the cost function. Thus, with the trivial LTL formula `true` (for which every trajectory is feasible, independently of the labeling on states), finding an optimal trajectory at least requires solving an infinite horizon optimal control problem for a nonlinear system. To proceed toward any solution for Problem 6, it is necessary to impose assumptions about the representation of trajectories. From there, particular cases of f and J can be treated.

5.3 The basic CE-LTL algorithm

The first proposed solution is a generalization of the basic CE motion planning algorithms introduced by Kobilarov [33]. The basic insight here is that the feasibility requirement of trajectories not being in collision with obstacles can be generalized to satisfaction of an arbitrary LTL formula. The case of avoiding obstacles and reaching a goal region is captured by the LTL formula $\Box \neg \text{Obstacle} \wedge \Diamond \text{Goal}$, provided a modification to the LTL semantics to permit the trajectory to end (thus produce a finite trace) upon reaching a state in Goal. LTL is interpreted on infinite strings, and thus a representation for trajectories is chosen that provides traces of infinite length and is amenable to model checking.

Before beginning, the major steps of the basic CE-LTL are listed in summary. Broadly interpreted, these are the basic steps characteristic of any procedure for stochastic optimization [57].

1. Initialize a sampling distribution.
2. Sample until N trajectories that satisfy φ are found.
3. Order these according to cost J.
4. Adjust the sampling distribution toward the ρN lowest cost trajectories.
5. Repeat until an appropriate termination criterion is met.

5.3.1 Brief introduction to the cross-entropy method

One way to find trajectories that satisfy an LTL formula and minimize an objective function, and thereby solve Problem 6, is to first find a probability distribution over trajectories that has support precisely on the optimal set. If this could be achieved, then obtaining an optimal trajectory is simply a matter of sampling from that distribution. While it may seem that this approach leads to a problem that is more difficult than the one that we originally wanted to solve, the probabilistic approach has the advantage that it more readily admits approximation and, in principle, the well-known methods for learning parametric distributions can be brought to bear in a general manner, instead of relying on special structure of enumerations of trajectories for a particular dynamical system.

In this chapter, the major steps of optimization essentially follow those of the cross-entropy method [17]. In this section a brief introduction to it is given in order to provide some motivation for Algorithm 6, which is stated later in the chapter.

The original motivation for the cross-entropy method is estimation of rare events. In the context of optimization, the "rare event" is the random sampling of values of the problem variables that achieve the minimum. More precisely, let θ be a random variable taking values in Θ, and let $J : \Theta \to \mathbb{R}$ be an objective function that maps values of θ to positive real numbers. Let $p(\cdot, \bar{v})$ be the optimal probability density function over Θ. For example, this could be a mixture of Gaussian (normal) distributions. Let $\gamma > 0$. The estimation problem is then to find the probability that θ randomly sampled according to $p(\cdot, \bar{v})$ is such that $J(\theta) \leq \gamma$, i.e., to find

$$l = P_{\bar{v}}(J(z) \leq \gamma) = \mathbb{E}_{\bar{v}} \left(I_{\{J(\theta) \leq \gamma\}} \right),$$

where $P_{\bar{v}}$ is the probability measure corresponding to the density $p(\cdot, \bar{v})$, and $\mathbb{E}_{\bar{v}}$ is the corresponding expectation. An approximation of this is

$$\hat{l} = \frac{1}{N} \sum_{i=1}^{N} I_{\{J(\theta_i) \leq \gamma\}} \frac{p(\theta_i, \bar{v})}{p(\theta_i, v)}$$

where $\theta_1, \ldots, \theta_N$ are independent and identically distributed samples from $p(\cdot, v)$, which is called the importance density. Since the desired distribution of optimal values is rare, $\hat{l}$ will practically be zero because samples will rarely occur in the support of the indicator function. This rarity is addressed in the cross-entropy method by beginning instead with an arbitrary initial density function parameter v_0 (practically chosen to represent any prior information) and a cost bound γ_1 that is sufficiently large such that $P_{v_0}(J(\theta) < \gamma_1)$ is not too small. Using this initial distribution, N samples $\theta_1, \ldots, \theta_N$ are produced and then the best (i.e., lowest cost in terms of J) among those are chosen and used to modify the probability density toward them, yielding a new density parameter v_1. Then the process repeats. This yields a sequence that converges in the sense of the Kullback-Leibler (KL) divergence (or the cross-entropy) to an optimal density parameter for importance sampling.

In some more detail, let $\rho < 1$ be a small scalar value, e.g., 0.1. Sample $\theta_1, \ldots, \theta_N$ from the distribution defined by $p(\cdot, v_0)$. The level γ_1 is obtained by sorting the sampled values so that $J(\theta_{s(1)}) \leq \cdots \leq J(\theta_{s(N)})$ for some permutation s of $\{1, \ldots, N\}$ and assigning $\gamma_1 := J(\theta_{s(\lceil \rho N \rceil)})$. The *elite set* $\mathcal{E}_1$ is the set of samples with cost at most γ_1, i.e., $\{\theta_i \mid J(\theta_i) \leq \gamma_1\}$, which has at least $\lceil \rho N \rceil$ elements by construction. The density parameter for the cost level γ_1 is then approximated by

$$v_1^* = \arg\max_{v \in \mathcal{V}} \frac{1}{N} \sum_{i=1}^{N} I_{\{J(\theta_i) \leq \hat{\gamma}_1\}} \ln p(\theta_i, v).$$

To avoid overfitting, the new parameters are taken to be a weighted sum of the best-fit (above) and the density parameters of the previous iteration, i.e.,

$$v_1 = \alpha v_1^* + (1 - \alpha) v_0,$$

where $0 < \alpha < 1$. These steps are repeated until samples meeting some termination criterion for the original optimization problem is met, e.g., if γ_j does not change much during several iterations.

5.3.2 Representation of trajectories

An infinite string $\sigma : \mathbb{N} \to 2^{\mathrm{AP}}$ is said to be *eventually periodic* if there exist τ, $T > 0$ such that for every $t > \tau$, $\sigma(t) = \sigma(t + T)$, in which case T is called the period, $\sigma_{[0,\tau]}$ is the prefix and $\sigma_{[\tau+1,\tau+T]}$ the periodic suffix. Another common notation to express infinite repetition of the finite suffix string after the prefix is $\sigma_{[0,\tau]} \left(\sigma_{[\tau+1,\tau+T]}\right)^{\omega} = \sigma$.

Let φ be an LTL formula in terms of the atomic propositions AP. The set of strings that satisfy φ is nonempty if and only if it contains a string that is eventually periodic. Motivated by this fact and practical considerations, all trajectories generated by the forthcoming algorithm are eventually periodic. This is restrictive in that there may be optimal trajectories satisfying an LTL formula that are not eventually periodic. The author is not aware of any examples at the time of writing, but one could begin with a line of irrational slope on the torus where states are labeled according to a uniform grid. It is an interesting direction of future work to find examples of this in practical scenarios.

Let $\mathbf{x} : [0, M] \to X$, $\mathbf{u} : [0, M] \to U$, where M is a positive integer, and let $\tau \geq 0$. Suppose that $\mathbf{x}(t + 1) = f(\mathbf{x}(t), \mathbf{u}(t))$ for $0 \leq t \leq M - 1$, and that $\mathbf{x}(\tau + 1) = f(\mathbf{x}(M), \mathbf{u}(M))$. The triple $(\mathbf{x}, \mathbf{u}, \tau)$ determines an eventually periodic trajectory $(\hat{\mathbf{x}}, \hat{\mathbf{u}})$ with period $T = M - \tau$ by taking

$$\hat{\mathbf{x}} = \mathbf{x}_{[0,\tau]} \left(\mathbf{x}_{[\tau+1,M]}\right)^{\omega}$$
$$\hat{\mathbf{u}} = \mathbf{u}_{[0,\tau]} \left(\mathbf{u}_{[\tau+1,M]}\right)^{\omega},$$

where the notation of infinite string repetition is used (cf. §2.1). The *trace* of $(\mathbf{x}, \mathbf{u}, \tau)$ is then defined as $\mathcal{L}((\mathbf{x}, \mathbf{u}, \tau)) = \mathcal{L}((\hat{\mathbf{x}}, \hat{\mathbf{u}}))$, and the *cost* as $J((\mathbf{x}, \mathbf{u}, \tau)) = J(\hat{\mathbf{x}}, \hat{\mathbf{u}})$. Because of this construction, triples $(\mathbf{x}, \mathbf{u}, \tau)$ satisfying the above properties will be regarded themselves as eventually periodic trajectories, without an explicit reference to $(\hat{\mathbf{x}}, \hat{\mathbf{u}})$.

To be a candidate solution, a pair of sequences $(\mathbf{x}, \mathbf{u})$ must be feasible in two respects. First, it must satisfy the dynamics (5.1), i.e., it must be a trajectory of the control system. Second, the corresponding sequence of labels $\mathcal{L}((\mathbf{x}, \mathbf{u}))$ must satisfy

the LTL formula φ. Directly sampling a trajectory, i.e., a set of values in X and U, will be practically impossible because the equality constraint that is the difference equation (5.1) corresponds to an event of measure zero. Therefore, sampling is instead from a parameter space, and each parameter vector uniquely determines a trajectory that is dynamically feasible by construction.

In more detail, let Θ be a manifold on which a parametric probability density function $p(\cdot, v)$ is defined. Suppose there is a subroutine GenTraj that is given and can produce trajectories that are eventually periodic from points in Θ, i.e.,

$$(\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta) = \textsc{GenTraj}(\theta).$$

The details of this subroutine depend on the dynamics under consideration. Details for the case of Dubins car and the single-integrator are given later in this chapter in §5.5. Two common approaches in practice are piecewise-constant control inputs, and waypoints in the state space. For the former, a trajectory is easily constructed by integrating (5.1) and then solving a two-point boundary value problem back to form a repeating suffix of the trajectory. For the latter, a sequence of boundary-value problems must be solved.

5.3.3 Deciding feasibility of trajectories

Given a sampled parameter vector θ, let $(\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)$ be the eventually periodic trajectory generated from it, as described in the previous section. Let T be the period. Then the trace is

$$\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) = \mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta))_{[0,\tau_\theta]} \left(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta))_{[\tau_\theta+1,\tau_\theta+T]}\right)^\omega,$$

which is an infinite string of 2^{AP}. Let φ be an LTL formula. The problem of deciding whether $\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta))$ satisfies φ is well-studied, and there are many algorithms for doing so. Like the trajectory construction from a parameter vector, this is another part of the algorithm for which an interchangeable subroutine is invoked. In

this section, several possibilities are described. Only the first, checking satisfaction by a deterministic Rabin automaton that recognizes φ, is considered in the numerical experiments presented in §5.5. The relaxation presented in §5.4.2 later in this chapter focuses on Büchi and Rabin automata, but the principles can be applied for ω-automata having other types of acceptance.

One of the most important approaches to checking whether a string satisfies an LTL formula is by constructing an ω-automaton equivalent to the LTL formula and then checking whether that string is in the language of (i.e., is accepted by) that automaton [62]. Equivalence is defined as equality of language, where the language of an LTL formula is the set of all strings satisfying it. For finite automata on finite strings, determinism does not affect the class of languages recognized. For ω-automata, which are defined for infinite strings, determinism can affect the expressiveness, notably for Büchi acceptance. Thus, in applying automata-based model checking in the present work, we must consider the exponential cost in size of deterministic automata compared to nondeterministic automata that recognize the same language, and this must be compared to the increased time complexity of checking whether a string is accepted by a nondeterministic automaton. In the sequel, feasibility of trajectories is checked using a deterministic Rabin automaton (defined next) because the exponential size complexity can be incurred ahead of time, and in trade, the time required for checking each trajectory is only that required to deterministically produce the corresponding path in the automaton (as a directed graph).

A deterministic Rabin automaton $\mathcal{A}$ is a tuple $(Q, q_0, \delta, \mathcal{F}, \Sigma)$, where Q is a finite set, $q_0 \in Q$, and $\delta : Q \times \Sigma \to Q$. $\mathcal{F} = \{(F_0, B_0), \ldots, (F_K, B_K)\}$. A *run* of $\mathcal{A}$ is a function $r : \mathbb{N} \to Q$ where $r(0) = q_0$ and for $t \geq 0$, there is some $a \in \Sigma$ such that $\delta(r(t), a) = r(t+1)$. An infinite string $\sigma : \mathbb{N} \to \Sigma$ is said to be accepted by $\mathcal{A}$ if there is some run $r : \mathbb{N} \to Q$ such that for $t \geq 0$, $\delta(r(t), \sigma(t)) = r(t+1)$, and there is some k such that $\mathrm{Inf}(r) \cap F_k \neq \emptyset$ and $\mathrm{Inf}(r) \cap B_k = \emptyset$, where

$$\mathrm{Inf}(r) = \{q \in Q \mid \forall t.\, \exists \tau > t.\, r(\tau) = q\}. \tag{5.2}$$

Algorithm 6 CE-LTL

1: INPUT: LTL formula φ, cost function J, initial state x_0, initial sampling distribution $p(\cdot, v_0)$, number of trajectories per iteration N, quantile $\rho < 1$, threshold cost γ
2: OUTPUT: parameters θ_* of best trajectory found
3: $j := 0$ //Iteration counter
4: **repeat**
5: **for all** $i = 1, \ldots, N$ **do**
6: **repeat**
7: $\theta_i \sim p(\cdot, v_j)$
8: $(\mathbf{x}_{\theta_i}, \mathbf{u}_{\theta_i}, \tau_{\theta_i}) = \textsc{GenTraj}(\theta_i)$
9: **until** $\mathcal{L}((\mathbf{x}_{\theta_i}, \mathbf{u}_{\theta_i}, \tau_{\theta_i})) \models \varphi$
10: **end for**
11: sort $\theta_1, \ldots, \theta_N$ according to cost, $J((\mathbf{x}_{\theta_1}, \mathbf{u}_{\theta_1}, \tau_{\theta_1})) \leq \cdots \leq J((\mathbf{x}_{\theta_N}, \mathbf{u}_{\theta_N}, \tau_{\theta_N}))$
12: $j := j + 1$
13: $v_j := \textsc{Update}(v_{j-1}, \theta_1, \ldots, \theta_{\lceil \rho N \rceil})$
14: **until** $J((\mathbf{x}_{\theta_1}, \mathbf{u}_{\theta_1}, \tau_{\theta_1})) < \gamma$
15: **return** θ_1 //Parameters of a trajectory with least cost

The set of strings accepted by $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

Since nondeterministinc Büchi automata are not used here, the details of deciding acceptance are not presented, but note that a search must be performed because there can be many possible paths for the same string (being nondeterministic).

5.3.4 Algorithm

The basic CE-LTL method is presented as Algorithm 6. Usage of the qualifier "basic" is motivated by clearly distinguishing Algorithm 6 from the extensions presented later in this chapter in §5.4. Several details to supplement reading of the algorithm:

- Line 7: The parameter vector θ_i is sampled according to the distribution with probability density function $p(\cdot, v_j)$. Note that besides finding parameters θ_* that provide a minimum cost trajectory, a distribution that is likely to sample optimal trajectory is also being estimated, consistent with the basic operation of the cross-entropy method (cf. §5.3.1).

- Line 8: Recall from §5.3.2 that trajectories are not sampled directly, but rather, parameter values from which trajectories can be constructed are sampled. The

details of GenTraj(θ_i) depend on the dynamical system.

- Line 9: Consult §5.3.3 for discussion about deciding feasibility of a trajectory.
- Line 13: Intuitively, adjust the parametric probability density function (by adjusting v_j) to make it more likely to sample the best $\lceil \rho N \rceil$ trajectories found during this iteration, while keeping some variance to avoid over-fitting. The Update subroutine is implemented as in §5.3.1. Other methods for learning a parametric density function could be applied here, e.g., EM (expectation-maximization).
- Line 14: The algorithm terminates when the minimum cost found is below a threshold. Two other common termination conditions are upon a fixed number of iterations and when the minimum cost has not changed much during several iterations. These and others are described in standard books on stochastic optimization, e.g., [57].

The part of the algorithm in which N feasible trajectories are sampled (lines 7–9) is trivially parallelizable, i.e., an arbitrary number of concurrent processes can be used to find the N trajectories and no coordination among them is needed. As an implementation detail, it is important to take care that pseudo-random number generators used in each thread have independent state, but this does not constrain the amount of concurrency. This capability is a very attractive feature of the algorithm (and the CE method in general) because randomly sampling trajectories that satisfy an LTL formula is challenging, and motivates the techniques developed in the next section.

5.4 Relaxations of the basic method

Without a good prior for the sampling distribution, it is a rare event to obtain a trajectory that satisfies an LTL formula. During the first several iterations, most time is passed sampling and discarding trajectories that are not feasible. Overcoming

this key difficulty of the initial conditions is thus well motivated for practical applications of CE-LTL planning. It is important to observe that this is not a special difficulty encountered for CE-LTL. Any stochastic optimization algorithm that seeks to randomly find sequences that satisfy an arbitrary LTL formula will encounter this difficulty. Though not explored more carefully here, this should not be surprising because the problem is essentially that of deciding whether certain trajectories exist in an arbitrary dynamical system, which is known to be intractable or undecidable [9]. As usual for such fundamental complexity barriers occurring for general statements of stochastic optimization problems, better performance can be achieved by exploiting special structure available for certain classes of specifications and dynamical systems. In this section, several relaxations are proposed that provide heuristics shown empirically to improve performance of CE-LTL.

5.4.1 Incrementally restrictive LTL formulae from templates

Let φ be an LTL formula as in the statement of Problem 6. The feasible set of trajectories over which an optimum is sought is described entirely by satisfaction of φ. Of course, the trajectories themselves must be dynamically feasible (i.e., satisfy the difference equation (5.1)), which is guaranteed by construction from the subroutine GenTraj for creating trajectories from parameter vectors (cf. §5.3.2). Since satisfaction of the LTL formula φ is challenging when constrained to search over dynamically feasible trajectories, even if simpler dynamics are considered, direct manipulation of φ is a good choice to increase the size of the feasible set. This increase can be trivially achieved by using the constant `true` formula instead. In that case, all trajectories are feasible (all strings satisfy `true` by definition; cf. §2.2). While sampling for the modified specification is (trivially) easier, we are still interested in solving the original problem, i.e., we want an optimal trajectory that satisfies φ. Thus something between the extremes of φ and `true` is desired.

The basic idea of the heuristic proposed in this section is to create a sequence of LTL formulae that converges to the original specification φ by beginning with an

(relatively) easy formula and then incrementally restricting it until the original LTL formula is obtained. Let $\varphi_K, \varphi_{K-1}, \varphi_{K-2}, \ldots, \varphi_0$ be LTL formulae such that $\varphi_0 = \varphi$ and

$$\mathcal{L}(\varphi_{k-1}) \subseteq \mathcal{L}(\varphi_k) \tag{5.3}$$

for $k \geq 1$. Thus, for any infinite string $\sigma : \mathbb{N} \to 2^{\mathrm{AP}}$, $\sigma \models \varphi_{k-1}$ implies that $\sigma \models \varphi_k$, but the converse may not hold. The notion of being restrictive is thus made precise here in terms of language containment. As we are in the setting of optimal control, satisfaction is by trajectories, so the sequence of LTL formulae provide that if $\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \models \varphi_l$ for some trajectory $(\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)$, then $\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \models \varphi_k$ for all $k \geq l$. The LTL formulae thus provide a sequence of feasible sets of trajectories, each being a subset of the previous, which implies that

$$P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \models \varphi_k) \geq P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \models \varphi_{k-1}),$$

where $P_v(\cdot)$ is the probability measure corresponding to the probability density function with parameter v. Using this observation, the heuristic proposed is to perform one iteration of the CE-LTL algorithm for φ_K and update the parametric probability density using the lowest cost feasible trajectories found (as part of that single iteration). Then perform the CE-LTL algorithm for φ_{K-1} initializing the density function with v found from the previous iteration (in which feasibility was with respect to φ_K). Repeat this until $\varphi_0 = \varphi$ (the original LTL formula) is obtained.

To employ this heuristic, a procedure for producing a sequence of LTL formulae that are related as in (5.3) must be presented. In motion planning for robotics, there are templates of important problems that provide a basis for this. For example, an LTL formula that captures the essential reach-avoid task is

$$\square \diamondsuit \psi_1 \wedge \square \diamondsuit \psi_2 \wedge \cdots \wedge \square \diamondsuit \psi_n \wedge \square \neg \phi, \tag{5.4}$$

where ϕ is satisfied precisely on obstacles in the workspace, and the other subformulae $\psi_1, \ldots, \psi_n$ correspond with subsets that are to be repeatedly visited. Among the

experiments described later in §5.5, each ψ_i is just an atomic proposition that labels an axis-aligned rectangle in $\mathbb{R}^2$ that is regarded as a goal, and ϕ is the formula $p_{n+1} \vee \cdots \vee p_{n+m}$, where each of $p_{n+1}, \ldots, p_{n+m}$ is an atomic proposition uniquely labeling an axis-aligned rectangle that is regarded as an obstacle.

Decomposing formula (5.4) into a sequence consistent with the relation (5.3) is achieved by first removing all subformulae except that of a single goal, then adding a second goal, etc., until all goals have been appended, and finally obstacles are included. Explicitly,

$$\begin{aligned}
\varphi_{n+1} &= \square \diamondsuit \psi_1 \\
\varphi_n &= \square \diamondsuit \psi_1 \wedge \square \diamondsuit \psi_2 \\
&\vdots \\
\varphi_1 &= \square \diamondsuit \psi_1 \wedge \square \diamondsuit \psi_2 \wedge \cdots \wedge \square \diamondsuit \psi_n \\
\varphi_0 &= \square \diamondsuit \psi_1 \wedge \square \diamondsuit \psi_2 \wedge \cdots \wedge \square \diamondsuit \psi_n \wedge \square \neg \phi,
\end{aligned}$$

where the final formula is the actual one with respect to which an optimal trajectory is desired.

5.4.2 Incrementally restrictive ω-automata

The basic idea of the previous section can be generalized to sequences of ω-automata that each recognize a more restrictive language, ending in an ω-automaton that recognizes precisely the actual desired LTL formula φ.

Let $\mathcal{A} = (Q, q_0, \delta, \mathcal{F}, \Sigma)$ be a deterministic Rabin automaton as defined in §5.3.3, with acceptance sets $\mathcal{F} = \{(F_0, B_0), \ldots, (F_J, B_J)\}$. A sequence of sets is constructed beginning with $\mathcal{F}_0 = \{(F_0^0, B_0^0), \ldots, (F_J^0, B_J^0)\} := \mathcal{F}$ and following the recursive iteration

$$F_j^{k+1} := \mathrm{Pre}(F_j^k) \cup F_j^k, \tag{5.5}$$

where Pre is defined by using edges available from δ with endpoints in Q (so, regarding $\mathcal{A}$ as a directed graph), and where $j \in \{0, \ldots, J\}$ is the index of the acceptance.

This recursion is guaranteed to terminate for all of the acceptance pairs $(F_k^j, B_k^j) \in \mathcal{F}$ because $\mathcal{A}$ has finitely many states. Let K_F be the smallest integer such that $F_j^{K_F+1} = \text{Pre}(F_j^{K_F}) \cup F_j^{K_F}$ for all j, i.e., it is an iteration of (5.5) sufficiently large to include fixed-points for all of the acceptance pairs. Notice that the B_j sets do not change in this part of the sequence. Thus continue the sequence by beginning with $k := K_F$ and applying the recursion

$$B_j^{k+1} := \text{Post}(B_j^k) \cap B_j^k, \tag{5.6}$$

where Post is defined by regarding transitions from δ as edges. Again fixed-points will necessarily be reached. Let K be the smallest integer such that $B_j^K = \text{Post}(B_j^K) \cap B_j^K$ for all j. Summarizing the above, a sequence of acceptance sets have been obtained: $\mathcal{F}^K, \mathcal{F}^{K-1}, \ldots, \mathcal{F}^0$. Denote by $\mathcal{A}_K, \mathcal{A}_{K-1}, \ldots, \mathcal{A}_0$ the Rabin automata accepting these, respectively. The final one, $\mathcal{A}_0$, recognizes precisely strings that satisfy φ and thus can be used to decide feasibility of trajectories for the original problem. Furthermore, it follows that

$$\mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{A}_1) \subseteq \cdots \subseteq \mathcal{L}(\mathcal{A}_K),$$

hence

$$\begin{aligned} P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \models \varphi) &= P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \in \mathcal{L}(\mathcal{A}_0)) \\ &\leq P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \in \mathcal{L}(\mathcal{A}_1)) \\ &\vdots \\ &\leq P_v(\mathcal{L}((\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)) \in \mathcal{L}(\mathcal{A}_K)). \end{aligned}$$

Therefore, having obtained a sequence of ω-automata that are incrementally more restrictive in the sense of language containment, we can perform one iteration of CE-LTL for each $k \geq 1$ using $\mathcal{A}_k$ to check feasibility of sampled trajectories. The parametric probability density is adjusted following the application of each automa-

ton, analogously to the sequence of LTL formulae proposed in §5.4.1. Observe that a Büchi automaton is equivalent to a Rabin automaton with one acceptance pair of the form $(F_0, \emptyset)$, and thus the preceding construction is applicable.

Note that performing the CE-LTL algorithm for satisfaction on intermediate Rabin automata solves a different (arguably, more general) problem than Problem 6 because the ω-automaton might not accept a language corresponding to any LTL formula. The issue is one of expressivity for languages of infinite strings and is potentially a subtle consideration. However, the ambition of this section is to propose a heuristic (relaxation) to improve convergence rates of CE-LTL in some cases, so using an automaton instead of an LTL formula for intermediate steps toward solving the optimal control Problem 6 is reasonable.

5.5 Numerical experiments

Provided that the feasible set is not empty, CE-LTL converges to a local minimum, but this is an asymptotic guarantee analogous to the probabilistic completeness of the motion planning algorithm RRT. Practically it is also very useful to know how quickly it will converge. Convergence rate is still an open question for CE-LTL, as it is for the CE method. Indeed, the situation is not any better for most stochastic optimization algorithms [57]. Furthermore, because of so-called "no free lunch" theorems, universally good performance cannot be achieved. Thus we are motivated to study performance empirically for particular cases. In this section, results are presented of experiments involving Dubins car and the single-integrator with LTL formulae of the reach-avoid template.

5.5.1 Dynamical systems and representations

The original paper by Dubins was not concerned explicitly with vehicles but rather optimal motion generally with constraints on curvature and constant velocity norm [19]. While many results in that paper are for arbitrary dimensional Euclidean spaces $\mathbb{R}^n$, the crucial results characterizing optimal curves are for the case of $n = 2$ (i.e.,

for planar motion). These have been of lasting importance in control theory because problems of steering vehicles that are nonholonomic are of fundamental interest, arising practically in cars and aerial vehicles. The basic unicycle model gives rise to variants with smooth steering as well as half-cars, trailers, etc. that are used for planning and control [37]. The Dubins car model is defined by the differential equation

$$\begin{aligned}\dot{x}_1(t) &= \cos x_3(t)\\ \dot{x}_2(t) &= \sin x_3(t)\\ \dot{x}_3(t) &= u(t),\end{aligned}$$

where t is a nonnegative real number and u is the control input (also referred to as turning rate). It is common to instead refer to the state variables as (x, y, θ) to emphasize that it is a point in $\mathbb{R}^2 \times S^1$, i.e., in SE(2). However, for consistency of notation in this chapter, $x(t) = (x_1(t), x_2(t), x_3(t))$ is used instead. Note that this is a differential equation where time $t \in \mathbb{R}$. A zero-order hold can be used to obtain a discrete-time system from integration. Explicitly, for constant input $\omega = 0$ during a duration of h seconds, we have

$$\begin{aligned}x_1(t+h) &= h\cos x_3(t) + x_1(t)\\ x_2(t+h) &= h\sin x_3(t) + x_2(t)\\ x_3(t+h) &= x_3(t),\end{aligned}$$

and for constant input $\omega \neq 0$,

$$\begin{aligned}x_1(t+h) &= \frac{1}{\omega}\left(\sin(x_3(t)+h\omega) - \sin x_3(t)\right) + x_1(t)\\ x_2(t+h) &= \frac{1}{\omega}\left(\cos x_3(t) - \cos(x_3(t)+h\omega)\right) + x_2(t)\\ x_3(t+h) &= h\omega + x_3(t).\end{aligned}$$

Recall from §5.3.2 that the space of trajectories over which optimization is performed is defined using parameters $\theta \in \Theta$. In Algorithm 6, the subroutine GenTraj(θ)

deterministically produces from the parameter vector θ the eventually periodic trajectory $(\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)$. Trajectory representation and generation for Dubins car are as follows. Let N be a positive integer. The parameter space $\Theta \subset \mathbb{R}^{2\times N}$ is of matrices defining piecewise constant inputs and durations, written explicitly as matrices of the form

$$\begin{pmatrix} T_1 & T_2 & \dots & T_N \\ \omega_1 & \omega_2 & \dots & \omega_N \end{pmatrix}, \tag{5.7}$$

where $T_1, \dots T_N \geq 0$ are nonnegative durations, and elements in the second row are turning rates, which are bounded absolutely by 1, i.e., $|\omega_j| \leq 1$. Though $\theta \in \Theta$ is a matrix, it is referred to as a vector to emphasize that Θ is contained in a vector space. Let θ be of the form (5.7). An eventually periodic trajectory $(\mathbf{x}_\theta, \mathbf{u}_\theta, \tau_\theta)$ is constructed from it as follows. Let m be a nonnegative integer that determines the resolution of the discretized trajectory. (It can be regarded as an implementation detail.) The finite state sequence $\mathbf{x}_\theta : [0, (N+1)(m+1)-1] \to \mathbb{R}^2 \times S^1$ is constructed in three steps. First fix the initial state $\mathbf{x}_\theta(0) = x_0$, which is given as part of this instance of Problem 6. Then, integrate for a total time of T_1 applying constant input ω_1 in $m+1$ steps (so, time increments of $\frac{T_1}{m+1}$), using the discrete-time equations given above, thus obtaining $\mathbf{x}_\theta(t)$ for $t \in \{0, \dots m+1\}$. Repeat for the remaining piecewise constant inputs, using the N columns of (5.7). The final parameter τ_θ determines the size of the suffix, i.e., the index at which $\mathbf{x}_\theta$ connects back to itself, which is $\tau_\theta + 1$ (cf. §5.3.2). While this could be considered to be a part of the trajectory parametrization and thus randomly sampled, it was not found empirically to be necessary. Indeed, the segment durations (the first row of (5.7)) are already being randomly sampled and thus the period in terms of physical time during the loop of the suffix is determined by the estimated probability density. In the implementation, the tie index $\tau_\theta + 1$ is fixed as 1/10 of the discrete length of generated trajectory before closing the loop (explicitly, it is $\left\lfloor \frac{N(m+1)+1}{10} \right\rfloor$). The trajectory construction is completed by solving the two-point boundary value problem between $\mathbf{x}_\theta(\tau_\theta + 1)$ and $\mathbf{x}_\theta(N(m+1))$, which is easily achieved by computing the several possibilities admitted as Dubins curves. The control input sequence that is part of the generated trajectory is obtained as part of

the steps described above, first being piecewise constant following (5.7) and then to realize a Dubins curve.

The second system considered is the single-integrator, which is defined by the ordinary differential equation

$$\dot{x}(t) = u(t),$$

where $x(t), u(t) \in \mathbb{R}^n$. This system essentially has no dynamics (i.e., is trivial) because velocity is directly controlled and motion in any direction is possible without constraint. Nonetheless it is interesting here because solving optimal control on it subject to an LTL formula specification emphasizes other aspects of Problem 6 besides dynamics. Single-integrators can still be of practical interest for slowly moving indoor mobile robots and in other situations as a rapid source of coarse trajectories for tracking. Trajectories of the single-integrator are parameterized (analogously to that of Dubins car) as a piecewise constant input but now allowing for arbitrarily many output dimensions, i.e., $\Theta \subset \mathbb{R}^{(n+1)\times N}$ and $\theta \in \Theta$ is of the form

$$\begin{pmatrix} T_1 & T_2 & \dots & T_N \\ u_{1,1} & u_{1,2} & \dots & u_{1,N} \\ & \vdots & & \\ u_{n,1} & u_{n,2} & \dots & u_{n,N} \end{pmatrix}, \tag{5.8}$$

where $T_1, \dots T_N \geq 0$ are nonnegative durations. The trajectory generation procedure GenTraj(θ) is entirely similar to that of Dubins car described above, except that the two-point boundary value problem to make the trajectory eventually periodic is trivial; it is merely linear interpolation between the two end-points.

Note that an alternative parameterization for all of the systems presented in this section is sampling states directly, i.e., θ is a finite sequence of states. Trajectories are then generated by solving a sequence of two-point boundary value problems between each column of θ.

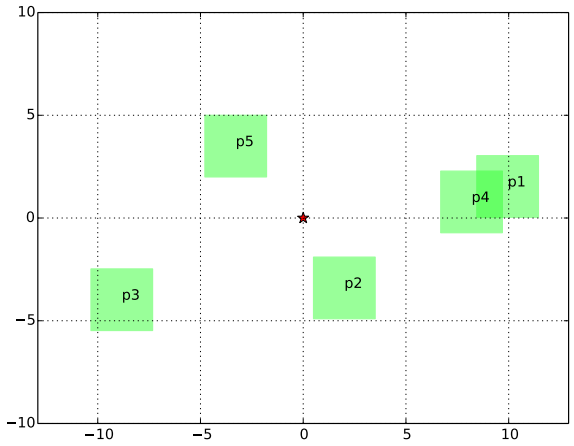

Figure 5.1: Example of a random problem instance for a workspace in $\mathbb{R}^2$. The task is to repeatedly visit the regions labeled p1, p2, and p3, while avoiding p4 and p5. The initial state is at the origin of the state space of the dynamical system, which has output to the origin in this plot, $(0, 0)$.

5.5.2 Comparisons among the basic method and relaxations

The basic algorithm (cf. §5.3.4) and the two heuristics based on relaxations of the original problem (cf. §5.4) were implemented and run in the trials described in this section. Details of the implementation are given in Appendix C. A summary of major features is that the implementation is in C++ and relies on the libraries Eigen for matrix representation and operations and Boost for multi-threading and pseudo-random number generation. Some supporting infrastructure is in Python, including C++ code generation for a function that checks acceptance by a deterministic Rabin automaton that recognizes a given LTL formula. Time required to generate the C++ code, compile it, and then dynamically link against the main program (without re-compiling the main program itself; i.e., only the acceptance function implementation is re-compiled per trial) is included in the listed run-times. Scripts providing visualization and statistics of results are in Python. For conciseness, the basic CE-LTL algorithm is referred to simply as `CE-LTL`, the relaxation described in §5.4.1 is referred to as `relax-formula-CE-LTL`, and that described in §5.4.2 is referred to as `relax-automata-CE-LTL`.

Let AP $= \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_{n+m}\}$ be a set of atomic propositions. Consider

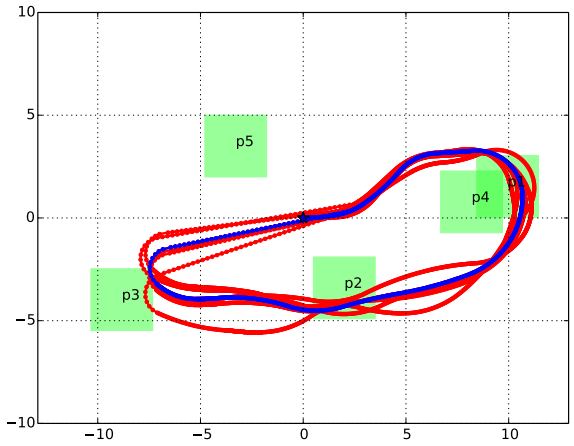

Figure 5.2: Several feasible trajectories of Dubins car for the problem instance depicted in Figure 5.1. The dark blue curve is the best found using `CE-LTL`. Notice that the trajectory is eventually periodic with a short prefix segment near the origin.

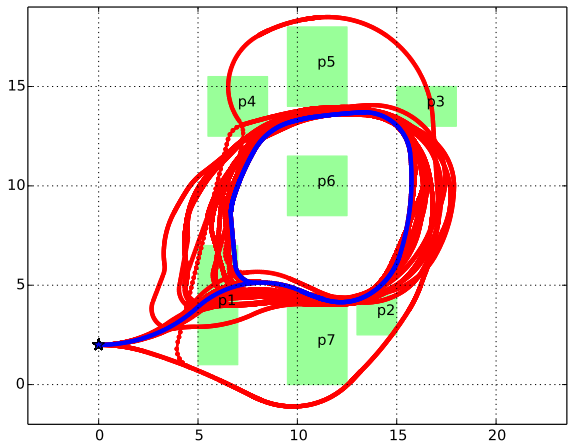

Figure 5.3: Best trajectories found during the 58 iterations of a single trial. The best (lowest cost) trajectory is depicted with dark blue. The sequence of costs are shown plotted in Figure 5.4. $\square\diamondsuit \mathrm{p1} \wedge \square\diamondsuit \mathrm{p2} \wedge \square\diamondsuit \mathrm{p3} \wedge \square\diamondsuit \mathrm{p4} \wedge \square\neg(\mathrm{p5} \vee \mathrm{p6} \vee \mathrm{p7})$ is the LTL formula.

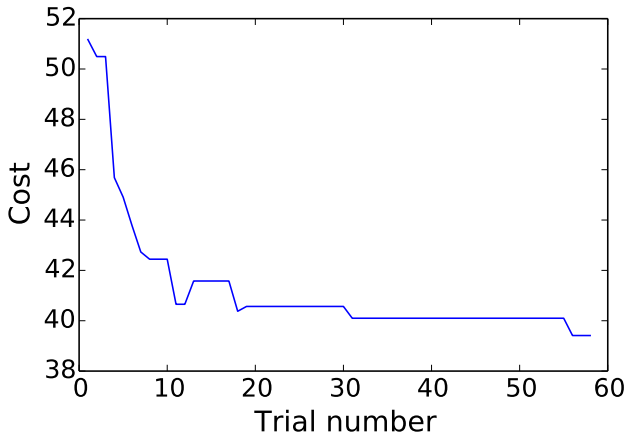

Figure 5.4: An example execution for the workspace shown in Figure 5.3. The cost for each iteration is the lowest among the feasible set found in that iteration.

the output space $\mathbb{R}^2$ in which axis-aligned rectangles are uniquely labeled, each with precisely one of the atomic propositions in AP. Thus there are |AP| rectangles total. These regions may be overlapping. An example is shown in Figure 5.1. Consider LTL formulae following the reach-avoid template of (5.4), for which the regions labeled $\{p_1\}, \ldots, \{p_n\}$ are to be repeatedly visited (i.e., infinitely often), and the regions labeled $\{p_{n+1}\}, \ldots, \{p_{n+m}\}$ are unsafe and must never be reached. An example solution from `CE-LTL` for Dubins car is shown in Figure 5.2. The cost function is path length, which is equal to trajectory duration (as a sum of the prefix and suffix parts) for Dubins car with unit speed as is used here. In addition to the lowest cost trajectory during the entire trial, several trajectories are shown that were of least cost among those found for particular iterations; recall that any such trajectory satisfies the LTL formula. An example is shown in Figure 5.3, and the cost of each iteration is plotted in Figure 5.4.

In all trials, the initial state is the origin. $n + m$ squares of side length 3 have centers randomly chosen in $[-10, 10]^2$. All algorithms are run with 8 simultaneous threads performing trajectories sampling. Convergence to a solution is declared if the change in lowest cost found between two iterations is less than 0.5.

While the running times provide some suggestion of absolute performance, caution is necessary because they are only for particular dynamical systems simulated

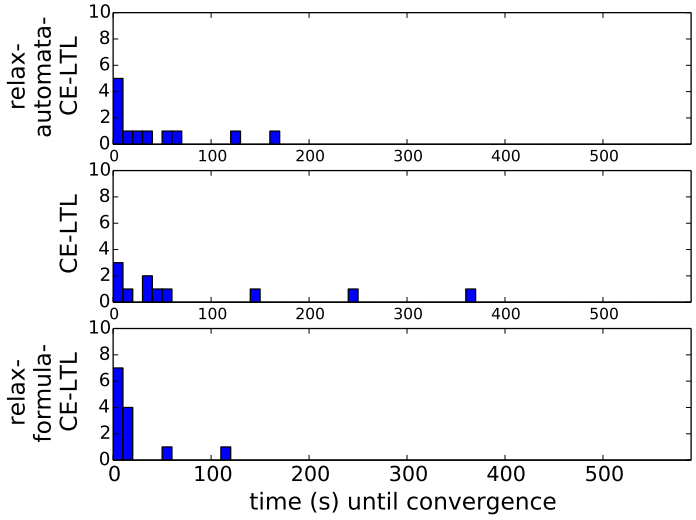

Figure 5.5: Histogram of run-times for 13 trials of the setting of Dubins car for tasks with 3 goal regions and 1 obstacle.

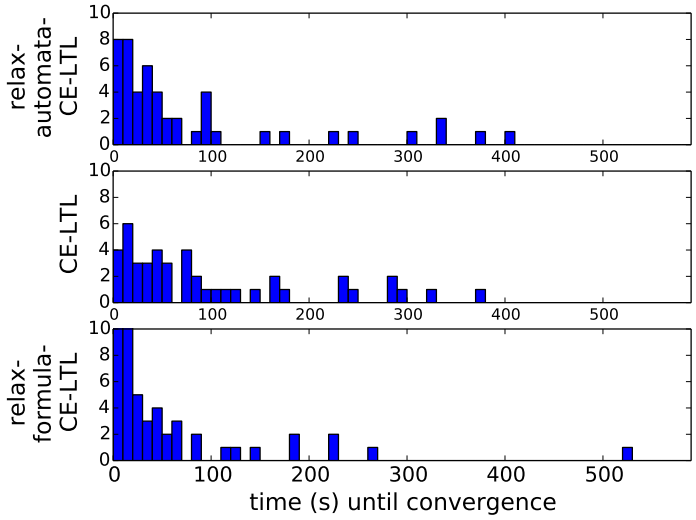

Figure 5.6: Histogram of run-times for 65 trials of the setting of Dubins car for tasks with 4 goal regions and 1 obstacle.

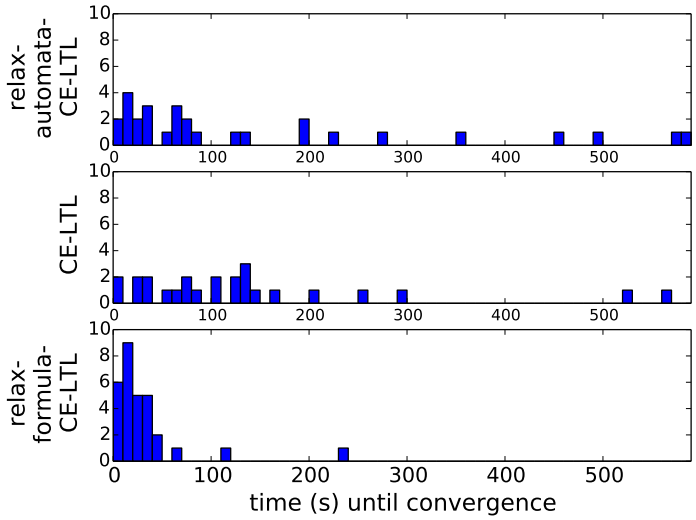

Figure 5.7: Histogram of run-times for 35 trials of the setting of Dubins car for tasks with 4 goal regions and 2 obstacles.

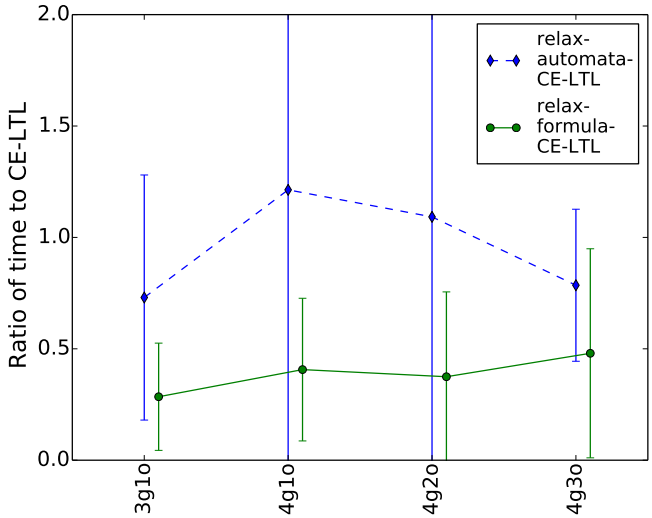

Figure 5.8: Ratios of times until convergence for `relax-formula-CE-LTL`, `relax-automata-CE-LTL` against the basic `CE-LTL`. Points are mean times, and error bars are 1 standard deviation. Data for 15 trials of the case of 4 goal regions and 3 obstacles are shown. For the other cases, counts of trials are as listed in the captions of Figures 5.5, 5.6, 5.7.

on a particular hardware configuration, and moreover, there are algorithm options like N, the number of trajectories to obtain per iteration CE-LTL, and ρ, which determines the size of the elite set (of lowest-cost trajectories; cf. §5.3.1). The tuning of these can affect performance and would need to be considered separately in practical deployments of the methods presented in this chapter, as is usual among applications of stochastic optimization algorithms [57]. However, the absolute times are indeed promising given the generality of Problem 6. Trends indicated by other aspects of the data, such as ratios of times, are not expected to depend on particular details of the examples and thus provide good indication of the relative strengths of the various methods `CE-LTL`, `relax-formula-CE-LTL`, `relax-automata-CE-LTL`. Comparison with related work is made in §5.5.3.

Except in obvious cases such as an obstacle region being randomly placed over the initial state, it is not easy to decide whether the feasible set is nonempty, i.e., whether there is some solution. All algorithms can be terminated before returning any solution if a user-configurable timeout occurs. While timeouts occurring suggest that the problem may not be feasible, we cannot reliably conclude this. Though this is made more difficult by the presence of an LTL formula, deciding when to terminate is generally difficult in stochastic optimization (because the global optimum is usually not known), and a variety of possibilities is worth considering in practice [57].

5.5.3 Comparison with related work

In this section, performances of the proposed CE-LTL method and relaxation-based heuristics are compared with an algorithm proposed by Karaman and Frazzoli for motion planning subject to deterministic μ-calculus specifications [30]. Note that there are several errors and missing details in their publication. Furthermore, a reference implementation was not available, so the author of this thesis created one. The algorithm with corrections is referred to here as `RRG*`. The implementations of the basic CE-LTL algorithm and relaxations are the same as was used in the previous section with the exception that random problem instances never have a point that

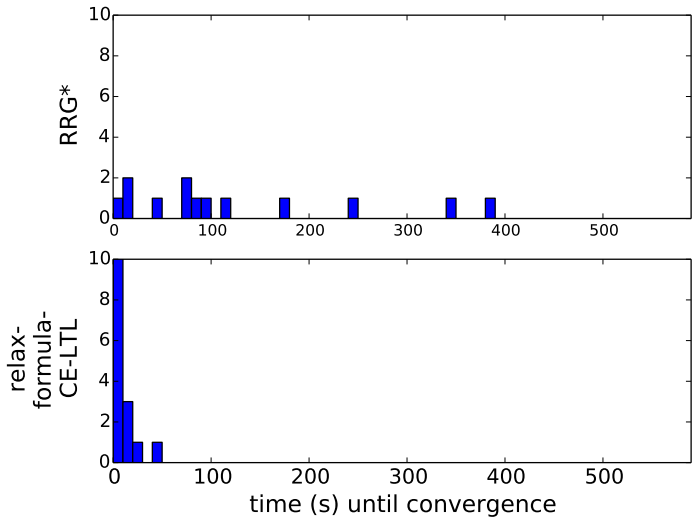

Figure 5.9: Comparison of convergence times between `relax-formula-CE-LTL` and `RRG*` in 16 trials for Dubins car in random trials of 3 goal regions and 1 obstacle. Each trial proceeds first with `relax-formula-CE-LTL` that has termination condition of 0.5 change in cost. The lowest cost found is then scaled by 1.5 to obtain an absolute threshold for `RRG*`, i.e., if a trajectory is found with at most that cost, then it terminates.

is in all goal regions simultaneously (i.e., the intersection of all goal regions must be empty). This is related to a constraint to have "distinct subsets" (mutually disjoint) as the labeled regions when applying `RRG*` as in the original papers [28, 30]. The current implementation of `RRG*` permits mutual intersections but when it includes all goals. Also consult Appendix C for details about the implementation.

In Figure 5.9, convergence times to the same cost are compared between `RRG*` and `relax-formula-CE-LTL`. The dynamical system is Dubins car, and the LTL formula follows the reach-avoid template described in the previous section. Each trial has 3 random goal regions and 1 static obstacle. A similar plot is shown in Figure 5.9 for the single-integrator and a formula having 4 goals and 2 static obstacles. While in the previous section the same termination condition was used for all algorithms, that approach was not appropriate here because `RRG*` frequently has sequences of iterations during which cost is only slightly improved, interleaved with jumps. Thus, with a termination condition based on change in best cost between iterations, `RRG*` may terminate early with a high cost (relative to that found by `CE-LTL`)

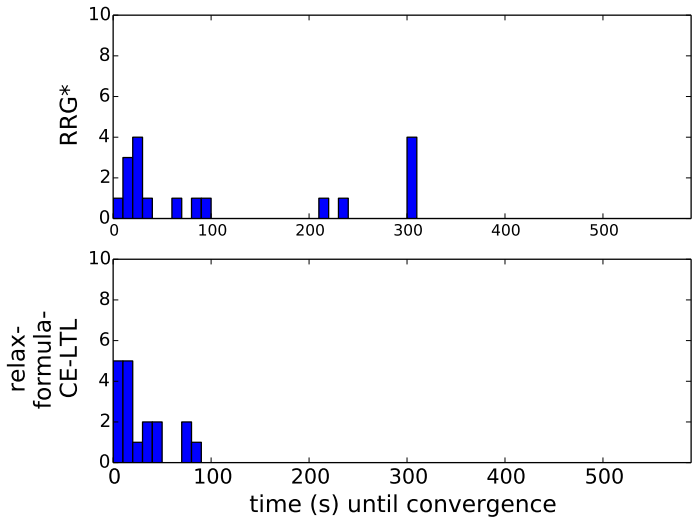

Figure 5.10: Comparison of convergence times between `relax-formula-CE-LTL` and `RRG*` in 18 trials for the single-integrator system in $\mathbb{R}^2$ in random trials of 4 goal regions and 2 obstacles. Each trial proceeds first with `relax-formula-CE-LTL` that has termination condition of 0.5 change in cost. The lowest cost found is then scaled by 1.5 to obtain an absolute threshold for `RRG*`, i.e., if a trajectory is found with at most that cost, then it terminates. A timeout of 300 s (5 minutes) was used for `RRG*`.

solution. To address this, each trial for comparison is performed in two parts. First `relax-formula-CE-LTL` is applied to the problem instance using a termination condition of change in best cost between iterations is less than 0.5. Upon completing, the optimum found is scaled by 1.5 (as a tolerance) to obtain an absolute threshold for `RRG*`.

Chapter 6

Conclusion

6.1 Summary and limitations

Though the foundations of robotics remain to be described, it is clear that a crucial aspect of any nontrivial robot architecture is planning and control. This may be considered in at least two respects: task strategy synthesis and execution, and trajectory generation and tracking. Each of these can be regarded as a kind of control synthesis, where the precise meaning of *controller* and *synthesis* depends on the particular problem considered. An architecture can simultaneously involve multiple notions of control synthesis and interaction among component controllers. In this thesis, two kinds of control synthesis are studied: discrete reactive synthesis, and open-loop trajectory generation. The former is formulated as a game with an adversarial environment (cf. §2.5) and has solutions that are expressed as finite-state machines, which are precisely defined as strategy automata in §3.4. The latter kind of control synthesis is based on a stochastic optimization algorithm with which low-cost open-loop trajectories are randomly sampled and constructed from piecewise constant inputs or solutions to two-point boundary-value problems, as presented in Chapter 5. These notions of control are distinct. They could be used together in different parts of the same architecture, or in certain applications one may be a more appropriate model than the other. In both cases, requirements are expressed using linear-time temporal logic. The main algorithms proposed in the thesis have in common that they are incremental. For the first kind of control synthesis, this is used to cope

with uncertainty as manifested in a changing GR(1) game. Methods are proposed for modifying strategies to be winning despite changes to reachability or liveness requirements of the game. For the second kind of control synthesis, the rare event of sampling a trajectory of a nonlinear system that satisfies an LTL formula is addressed by proposing two relaxations that involve sequences of incrementally restrictive LTL formulae or ω-automata that converge on the desired LTL formula.

Considerations about completeness, limitations, and practical applications are given in earlier chapters where relevant. Several salient issues are summarized here. For the work presented in Chapters 3 and 4 to be applicable to systems with state spaces of infinite cardinality, a discrete abstraction must be given. Presenting one for large practical systems is potentially a challenging first step before the presented methods for control synthesis can be applied. Deciding realizability can require a large amount of time (approaching that of global re-synthesis) and thus, it is not done as part of the algorithms presented in Chapters 3 and 4. As such, it may be discovered that patching is not feasible after attempts at many different sizes of local reachability games, leading to an overall computation time that is greater than if realizability had been checked directly. This seems unavoidable unless special structure in certain games is exploited to predict whether patching or global re-synthesis will succeed. As a topic of future work, current performance could be improved by reusing intermediate values obtained when solving reachability games for a particular neighborhood. The intuition is that attempting patching for monotonically larger sets of states (ordered by the subset relation) may correspond to a sequence of reachability games (ordered by substrategy). A key limitation of the treatment in Chapter 5 is that it does not include noise, disturbances, or other models of uncertainty about the dynamics. Nonetheless, for a practical deployment, trajectory tracking is a good option, and errors in tracking can be accounted for when ensuring feasibility with respect to the LTL formula by contracting or expanding labeled regions. The above mentioned limitations are motivations for future work because it is not known yet whether they are manifestations of fundamental barriers.

6.2 Future work

The methods and results of Chapters 3 and 4 are entirely for GR(1) games, which capture many sophisticated behaviors that are of both theoretical and practical interest. However, most of the development in this thesis is in terms of or can be reduced to intermediate sets from the fixed-point computations performed during synthesis. The salient structure seems to be the inner μ-ν alternation (cf. §2.4), and as such, future work is motivated toward extension of definitions like reach annotation and algorithms that use it to a fragment of μ-calculus that is larger than GR(1). Relatedly, the question of relevance to reactive synthesis for specification languages with distinct semantics, like metric temporal logic (MTL), is open.

The incremental modifications to strategy automata presented in Chapter 3 are performed around nodes that require changes without considering the impact of the change with respect to some cost function. As such, while computation time may be accordingly small, the patched strategy may cause trajectories that are correct but high-cost. Similarly, static obstacles may be added or removed from the workspace of the robot during long-running missions, which would imply that states can become unreachable and eventually reachable again. Because the patches on a strategy automaton are not specially marked (i.e., the result of patching is just another strategy automaton), and because newly reachable controlled states do not affect the strategy (cf. §3.8), incremental synthesis for a long-running controller could accumulate degradation of the original strategy as local avoidance behaviors are added but not later removed, despite becoming unnecessary. Incorporating some notion of cost minimization into the patching process and tracking and pruning patches that become unnecessary are topics of future work.

A separate method was presented in §3.10 for the case of changes to reachability that affect certain goal states. An alternative method for this case is obtained using the algorithms of Chapter 4 by removing and then adding back the affected system liveness conditions. Future work will elaborate this alternative and include a detailed analysis of and comparison with the method presented in §3.10.

As should be apparent from discussion in §5.3.3 and §5.4.2, deciding feasibility in the basic CE-LTL algorithm can be regarded as a subroutine that is readily replaced for specification languages besides LTL. For example, the treatment of Chapter 5 is essentially unchanged for MTL because sampled trajectories already have timing information. With an appropriate cost function, temporal logics that represent robustness of satisfaction are also applicable.

An important program of research is to identify useful and tractable (or less intractable) fragments of LTL and other temporal logics, as well as to propose specification languages that have distinct expressivity from existing ones. Some of these languages and fragments are defined using templates, e.g., GR(1). An ongoing direction of future work is to explore decompositions of these templates as a relaxation method for CE-LTL, analogous to the treatment of reach-avoid templates presented in §5.4.1. Exploring these relaxations can occur empirically by trying different decomposition patterns in an ad hoc fashion. Decompositions could also be informed by the dynamics and relative positions of labeled regions in the output space. For example, goals that are progressively farther from the initial state could be incrementally added, motivated by the expectation that that order corresponds to a good (low-cost) set of feasible trajectories.

Appendix A

Time semantics for two-player games

This appendix is primarily intended as a supplement to Chapters 3 and 4 by providing additional background material on notions of time and game graphs.

In the introduction to GR(1) games of §2.5, turn-taking is said to follow a Mealy time semantics. The basic idea is that the output depends on the current state and the input. This is in contrast to so-called Moore time semantics, where the output can depend only on the current state. The names refer to the authors of papers that considered finite-state machines with these respective notions of dependency [47, 48]. The definition of strategy automaton in §3.4 differs slightly from others by including labels of entire states on the nodes and by lacking of explicit sets of "inputs" and "outputs." Nonetheless, there is equivalence, which should be apparent if, for each node v of a strategy automaton, the incoming edges are modified to be labeled by an output $L(v) \cap \mathrm{AP}^{\mathrm{sys}}$. Since the edges (defined by the transition function δ) are already dependent on the input, i.e., the subset of $\mathrm{AP}^{\mathrm{env}}$ provided by the adversarial environment, we thus obtain a finite-state machine wherein inputs are read and outputs are given in response during the transitions. With this modification, the labels of nodes (provided by L) can be dropped. This still follows the Mealy time semantics because the output depends on the node from which the edge originates as well as the input.

The graph introduced in §3.2 is referred to as a game graph to emphasize that

one following the standard definition can be easily obtained. Before showing this, the standard definition is introduced. A *game graph* $\mathcal{G}$ is a graph $(V_0 \cup V_1, E)$, where V_0 and V_1 are disjoint (i.e., $V_0 \cap V_1 = \emptyset$), together with a set $\text{Win} \subseteq V^\omega$ of infinite plays, where an *infinite play* is an infinite string on $V = V_0 \cup V_1$ that is consistent with the edge set, i.e., for every $p \in \text{Win}$, $(p(t), p(t+1)) \in E$ for $t \geq 0$. A *strategy* for Player 0 is a partial function $f_0 : V^+ V_0 \to V$ that maps finite plays to some vertex. A play p is said to *conform* to the Player 0 strategy f_0 if for every t such that $p(t) \in V_0$, $p(t+1) = f_0(p(t))$. Intuitively, a play of $\mathcal{G}$ proceeds according to the motion of a token. If the token is on a vertex v in V_0, then Player 0 selects the next vertex on which to place the token among the successers (according to E) of v. Otherwise (i.e., when the token is on a vertex in V_1), Player 1 selects the next successor. A synthesis problem can then be posed as follows: let $v \in V$. Find a strategy f for Player 0 such that for every play p with $p(0) = v$ and conforming to f, $p \in \text{Win}$. Notice that no constraint is placed on transitions from vertices in V_1. We could instead introduce Player 1 strategies and then pose the problem against the presence of any Player 1 strategy, motivating description of it as adversarial. The set Win may be expressed using various winning conditions. E.g., an important class is parity games, in which a function $c : V \to \mathbb{N}$ is given, from which winning is defined as

$$\text{Win} = \{p \mid (\max \text{Inf}(c \circ p)) = 0 \mod 2\},$$

where Inf is defined in (5.2) in §5.3.3. Intuitively the winning condition requires that the maximum of the repeatedly occurring c-values is even, i.e., is divisible by 2.

The equivalence is shown constructively as follows. Let φ be a GR(1) formula as in (2.3), where the set of uncontrolled (environment) atomic propositions is AP^{env} and the set of controlled (system) atomic propositions is AP^{sys}. Let $G_\varphi = (\Sigma, E_\varphi^{\text{env}}, E_\varphi^{\text{sys}})$, where $\Sigma = 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}$, be the graph associated with φ as defined in §3.2. Let $V_0 \subseteq 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}} \times 2^{\text{AP}^{\text{env}}}$, $V_1 \subseteq 2^{\text{AP}^{\text{env}} \cup \text{AP}^{\text{sys}}}$, and $E \subseteq (V_0 \cup V_1)^2$ be the smallest sets such that

1. $\iota \in V_1$,

2. if $x \in 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$, $e \in E_\varphi^{\mathrm{env}}(x)$, then $(x,(x,e)) \in E$, and

3. if $x \in 2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$, $e \in E_\varphi^{\mathrm{env}}(x)$, and $y \in E_\varphi^{\mathrm{sys}}(x,e)$, then $((x,e),y) \in E$.

For a play $p : \mathbb{N} \to V_0 \cup V_1$ with $p(0) = \iota$, let the associated trace be

$$\mathcal{L}(p)(t) = p(2t) \qquad \text{for } t \geq 0.$$

It follows from the definitions of E and V_1 that $\mathcal{L}(p)$ is a string on $2^{\mathrm{AP}^{\mathrm{env}} \cup \mathrm{AP}^{\mathrm{sys}}}$. Thus, we can finally take Win to be all plays p with $p(0) = \iota$ and such that $\mathcal{L}(p) \models \varphi$.

Remark 21. $(V_0 \cup V_1, E)$ is bipartite. Every play of the GR(1) game (φ, ι) satisfying the transition rules ρ^{env} and ρ^{sys} corresponds to a walk on $\mathcal{G}$.

Appendix B

Probability theory

This appendix supplements Chapter 5, which presents novel stochastic optimization algorithms for control subject to LTL specifications. The contributions build on the cross-entropy method, which is briefly introduced in §5.3.1.

The probability density function of the normal distribution with mean $\mu \in \mathbb{R}^d$ and covariance matrix Q is

$$f(x, v) = \frac{1}{(2\pi)^{\frac{d}{2}} \det(Q)^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T Q^{-1}(x-\mu)\right),$$

where $v = (\mu, Q)$ is the parameter vector, which is thus written to be consistent with the notation used in Chapter 5 and to emphasize that we only need some parametric density function, not necessarily that of the normal distribution. A Gaussian mixture model is a weighted combination of normal distributions, so using f above, it would be $g(x, v) = \alpha_1 f(x, v_1) + \cdots \alpha_k f(x, v_k)$, where the coefficients (weights) are nonnegative and $\alpha_1 + \cdots + \alpha_k = 1$, and the density parameter is a tuple containing the component parameters together with coefficients. The experiments reported in §5.5 use a normal distribution, but the same implementation already supports using a Gaussian mixture model instead.

In each iteration of CE-LTL (and the CE method in general), the density parameter is adjusted "toward" the elite set, i.e., the $\lceil \rho N \rceil$ lowest cost trajectories found in that iteration (all of which are feasible, i.e., satisfy the LTL formula). While a general statement for this in terms of the parameter space $\mathcal{V}$ is provided in §5.3.1,

the case of normal density functions is provided here explicitly. Let $N_e = \lceil \rho N \rceil$, and let $\theta_1, \theta_2, \ldots, \theta_{N_e}$ be the sampled trajectory parameter values of lowest cost (possibly after re-ordering of subscripts). (Beware that "parameter" can by now be referring to three different objects: the trajectory parameters, the density function parameters, or the algorithm parameters, e.g., N. Which is intended should be clear from context.) Then, the best-fit density parameter is

$$\mu^* = \frac{1}{N_e} \sum_{j=1}^{N_e} x_j$$

$$Q^* = \frac{1}{N_e} \sum_{j=1}^{N_e} (x_j - \mu^*)(x_j - \mu^*)^T.$$

Appendix C

Implementation details

C.1 Introduction

The ambition of this appendix is to provide details about the implementations on which results are reported. These are mostly tangential to the main work of the thesis and so are better placed outside the main text. In summary, all dependencies are free, open source software, as are the implementations used to produce experiment results in this thesis. However, at the time of writing, several of the implementations of particular methods presented here have not been released yet, especially in the case of Chapter 5 and in particular the implementation of `RRG*` that is used in §5.5.3. The authors of the original papers about the related work [28, 30] are being consulted before releasing the new reference implementation, to allow discussion about the corrections and elaboration of details that were made while creating it.

C.2 Incremental control for GR(1) games

The algorithms presented in Chapters 3 and 4 are provided through the application-programming interface (API) of gr1c (`http://scottman.net/2012/gr1c`) and via the command-line program `gr1c patch`. Specifically, goal appending can be achieved using

```
gr1c patch -f PHI [...]
```

where `PHI` is an arbitrary state formula (i.e., without temporal operators), which is parsed using the same syntax as for GR(1) specifications. Also, the switch `-m` can be included in the above command to indicate that some variables are integer-valued and can be included in the computation of a norm (regarding the variables as elements in a vector), from which the Dist function is obtained; otherwise Dist is taken to be constant. Goal removal can be achieved using `gr1c patch -r i`, where `i` is the index, corresponding to removal of the system liveness (goal) subformula ψ_i^{sys} (following the statement of Problem 5). As is the case of the theoretical results, the implementation is entirely general in the context of GR(1) games. Note that gr1c is not restricted to atomic propositions, which can be regarded as Boolean (or binary) valued variables. Integer domains are also available. A manner of obtaining them from atomic propositions is described in §C.2.2.

The expression of changes to reachability is somewhat more complicated than changes to liveness. The approach taken in gr1c is to define a separate file type known as an "edge set change file." These files consist of two parts: first a list of states considered to be in the neighborhood N (cf. §3.8 and §3.9); and second, a sequence of `restrict`, `relax`, or `blocksys` commands that . As in other files for working with gr1c and following the convention of Python and UNIX shell scripts, comments begin with `#`. For example, if there are two atomic propositions p and q, and we have declared that $N = \{\emptyset, \{q\}, \{p, q\}\}$, and we wish to change the game graph by removing the controlled edge from $\emptyset$ to $\{q\}$, then the file would be

```
0 0
0 1
1 1
restrict 0 0 0 1
```

The experiments on gridworlds reported in §3.12 and §4.5 use TuLP, the temporal logic planning toolbox (`http://tulip-control.org`) for gridworld representation and for profiling that is used to obtain the synthesis reported synthesis times. TuLiP has an interface to gr1c, which indeed is the default GR(1) solver used by it.

C.2.1 Code for an example

The game of Example 2 in §3.4 is expressed by the following `gr1c` specification.

```
# gr1c -n ONE_SIDE_INIT strategyautillust.spc

ENV: door_open door_reached;
SYS: goto_door;

ENVGOAL: []<>(goto_door -> door_reached);

SYSINIT: True;
SYSTRANS:
  [](door_open -> goto_door')
& []((goto_door & !door_reached) -> goto_door');
SYSGOAL: []<>door_reached;
```

C.2.2 Representing variables of other types as atomic propositions

Throughout the thesis, properties are expressed and checked in terms of atomic propositions, e.g., using the propositional linear-time temporal logic (cf. §2.2). Atomic propositions can be regarded as Boolean variables taking the values `true` or `false`. This restriction of attention is motivated primarily by providing a concise treatment of the theory. However, it is without loss of generality because any variable that takes values on a finite domain can be encoded using Boolean variables. The construction is sketched here for the case of variables that can be assigned integers. Let x be an integer variable with domain $\{0, 1, \ldots n\}$, where $n \geq 1$. Let

$$k = \lceil \log_2(n) \rceil$$

and create k atomic propositions $x_0^{\text{bit}}, x_1^{\text{bit}}, \ldots, x_{k-1}^{\text{bit}}$. As the names suggest, these are interpreted as bits encoding the value of x. E.g., $x = 5$ (supposing that $n \geq 5$) would be represented by the state $\{x_0^{\text{bit}}, x_2^{\text{bit}}\}$. Since n might not be a power of 2, several states need to be precluded in order for the domain of x to be correctly represented. In GR(1) formulae, this is achieved by adding restrictive subformulae to ρ^{sys} if $n < k$,

$$\square \left(\neg \chi_{x=n+1} \wedge \neg \chi_{x=n+2} \wedge \cdots \wedge \neg \chi_{x=k} \right),$$

where $\chi_{x=k}$ is the characteristic formula for the state of $x = k$, which is actually a conjunction of the atomic propositions $x_0^{\text{bit}}, x_1^{\text{bit}}, \ldots, x_{k-1}^{\text{bit}}$, some negated; e.g., if $k = 3$,

$$\chi_{x=5} = x_0^{\text{bit}} \wedge \neg x_1^{\text{bit}} \wedge x_2^{\text{bit}}.$$

C.3 CE-LTL and relaxations

The implementation of the CE-LTL algorithm and relaxations of it that are presented in Chapter 5 (and used in the experiments described in §5.5) is in C++, along with some infrastructure in Python, and builds on the GCOP library by Marin Kobilarov, which provides a basic implementation of the cross-entropy method along with several dynamical systems and manifolds, e.g., multi-link bodies and SE(3). However, the single-integrator and Dubins car, including generation of Dubins curves to create periodicity in sampled trajectories, are implemented de novo. Matrix representation and operations are achieved with Eigen `http://eigen.tuxfamily.org`. The Boost libraries (`http://www.boost.org`) for multithreading and pseudo-random number generation are used. Algorithm timeouts are implemented by registering a POSIX alarm; it is hard interruption, i.e., the iteration in progress is not permitted to finish, which is motivated because in the case of the sampling part of the CE-LTL algorithm, an arbitrarily large amount of time could be required. Observe that a nonempty feasible set ensures that the iteration will finish with probability one, but no general rate is known.

Checking of feasibility of trajectories with respect to an LTL formula is achieved using a deterministic Rabin automaton (cf. §5.3.3). This is obtained using LTL2DSTAR (`http://www.ltl2dstar.de`) [32]. The output of LTL2DSTAR is in a tool-specific format. It is parsed by a Python program created as part of the CE-LTL implementation that generates C++ for checking acceptance of trajectories that are eventually periodic. The C++ code for the dynamical systems and the basic CE-LTL algorithm is compiled once. Acceptance criteria (i.e., the LTL formula with respect to which satisfaction is checked) are changed by separately compiling the generated C++ code as a dynamically linked library, to which the main programs link. During initial compilation, they are linked against a vacuous acceptance-checking function, which prints a warning message if accidentally used.

Tools for visualization and statistics are in Python using standard scientific Python packages including NumPy, Matplotlib, and NetworkX.

The relaxation heuristics described in §5.4 use the same external tool (LTL2DSTAR) that produces deterministic Rabin automata (DRA) from LTL formulae, and the same routine for C++ code generation is used with the following additional details. The method described as relaxation of the recognizing ω-automaton (cf. §5.4.2)) is implemented by generating the sequence of Rabin automata from the initial one produced by LTL2DSTAR. These are assigned levels that correspond to the subscript numbers of $\mathcal{F}_0, \mathcal{F}_1, \ldots$ presented in §5.4.2. An acceptance-checking function in C++ is then automatically generated for each DRA and wrapped in a namespace dedicated to that automaton. The client code selects a particular acceptance function by calling a mutex function (that is also automatically generated) that in turn calls the appropriate acceptance-checker given an integer level. Using level 0 provides satisfaction checking for the original LTL formula.

The method described in §5.4.1 as relaxing based on templates of LTL formulae proceeds in a similar fashion, except that each DRA level is produced from an LTL formula. Again, a C++ mutex function is automatically generated to switch among available acceptance criteria.

Bibliography

[1] R. Alur, T. Dang, and F. Ivančić. Counter-example guided predicate abstraction of hybrid systems. In *Proceedings of TACAS*, 2003.

[2] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, July 2000.

[3] K. J. Åström and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[5] M. Barbeau, F. Kabanza, and R. St.-Denis. A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. *IEEE Transactions on Automatic Control*, 43(11):1543–1559, November 1998.

[6] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas. Symbolic planning and control of robot motion: Finding the missing pieces of current methods and ideas. *IEEE Robotics & Automation Magazine*, pages 61–70, March 2007.

[7] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann. Robustness in the presence of liveness. In *Proceedings of Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 410–424. Springer Berlin / Heidelberg, 2010.

[8] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78:911–938, May 2012.

[9] M. S. Branicky. Universal computation and other capabilities of hybrid and continuous dynamical systems. *Theoretical Computer Science*, 138:67–100, 1995.

[10] M. S. Branicky. Multiple lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):475–482, April 1998.

[11] M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE Transactions on Automatic Control*, 43(1):31–45, January 1998.

[12] R. W. Brockett. Formal languages for motion description and map making. In *Robotics, Proceedings of Symposia in Applied Mathematics*, volume 41, pages 181–193. American Mathematical Society, 1990.

[13] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[14] A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, Stockholm, August 1962.

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

[16] D. C. Conner, H. Kress-Gazit, H. Choset, A. A. Rizzi, and G. J. Pappas. Valet parking without a valet. In *Proceedings of the 2007 IEEE/RSJ Int'l Conference on Intelligent Robots and Systems (IROS)*, pages 572–577, 2007.

[17] P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134:19–67, 2005.

[18] J. C. Doyle, B. A. Francis, and A. Tannenbaum. *Feedback control theory*. Macmillan Pub. Co., 1992.

[19] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497–516, July 1957.

[20] E. A. Emerson. *Handbook of theoretical computer science (vol. B): formal models and semantics*, chapter Temporal and modal logic, pages 995–1072. MIT Press, 1990.

[21] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258:491–522, 2001.

[22] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 3–18, Warsaw, Poland, June 1995.

[23] M. Guo, K. H. Johansson, and D. V. Dimarogonas. Rivising motion planning under linear temporal logic specifications in partially known workspaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 5010–5017, Karlsruhe, Germany, May 2013.

[24] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly ltl model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 191–205. Springer-Verlag Berlin Heidelberg, 2005.

[25] T. A. Henzinger. The theory of hybrid automata. In *Proc. of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[27] R. Isaacs. *Differential Games*. Wiley, 1965.

[28] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *Proceedings of the 48th IEEE Conference on Decision and Control (CDC)*, pages 2222–2229, Shanghai, China, 2009.

[29] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, 30(7):846–894, 2011.

[30] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning with deterministic μ-calculus specifications. In *Proc. of the American Control Conference (ACC)*, pages 735–742, Montréal, Canada, June 2012.

[31] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Information and Computation*, 200:35–61, 2005.

[32] J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363:182–195, 2006.

[33] M. Kobilarov. Cross-entropy motion planning. *Int. J. of Robotics Research*, 31:855–871, 2012.

[34] S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14 (NIPS)*, 2001.

[35] G. Lafferriere, G. J. Pappas, and S. Sastry. O-minimal hybrid systems. *Mathematics of Control, Signals, and Systems*, 13:1–21, 2000.

[36] S. M. LaValle. Rapidly-exploring random trees: A new tool for path-planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, October 1998.

[37] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[38] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, May 2001.

[39] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*, 2005.

[40] J. Liu and N. Ozay. Abstraction, discretization, and robustness in temporal logic control of dynamical systems. In *Proc. of Hybrid Systems: Computation and Control*, pages 293–302, April 2014.

[41] S. C. Livingston and R. M. Murray. Hot-swapping robot task goals in reactive formal synthesis. In *Proceedings of the IEEE 53rd Annual Conference on Decision and Control (CDC)*, pages 101–107, Los Angeles, CA, USA, December 2014.

[42] S. C. Livingston, R. M. Murray, and J. W. Burdick. Backtracking temporal logic synthesis for uncertain environments. In *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5163–5170, Saint Paul, Minnesota, USA, May 2012.

[43] S. C. Livingston, P. Prabhakar, A. B. Jose, and R. M. Murray. Patching task-level robot controllers based on a local μ-calculus formula. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4573–4580, Karlsruhe, Germany, May 2013.

[44] S. C. Livingston, E. M. Wolff, and R. M. Murray. Cross-entropy temporal logic motion planning. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 269–278, Seattle, WA, USA, April 2015.

[45] V. Manikonda, P. Krishnaprasad, and J. Hendler. A model description language and a hybrid architecture for motion planning with nonholonomic robots. In *Proceedings of the IEEE Int'l Conference on Robotics and Automation (ICRA)*, pages 2021–2028, 1995.

[46] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *(PODC '90) Proceedings of the ninth annual ACM Symposium on Principles of Distributed Computing*, pages 377–408, 1990.

[47] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal (BSTJ)*, 34(5):1045–1079, September 1955.

[48] E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34 in Annals of Mathematical Studies, pages 129–153. Princeton University Press, Princeton, New Jersey, USA, 1956.

[49] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, November 1977.

[50] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[51] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[52] J. H. Reif. Complexity of the mover's problem and generalizations. In *IEEE 20th Annual Symposium on Foundations of Computer Science*, pages 421–427, October 1979.

[53] S. Sarid, B. Xu, and H. Kress-Gazit. Guaranteeing high-level behaviors while exploring partially known maps. In *Proceedings of Robotics: Science and Systems*, Sydney, Australia, July 2012.

[54] S. Schewe. *Synthesis of Distributed Systems*. PhD thesis, Universität des Saarlandes, 2008.

[55] K. Schneider. *Verification of Reactive Systems: formal methods and algorithms*. Springer, 2004.

[56] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin / Heidelberg, 2005.

[57] J. C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, Inc., 2003.

[58] P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[59] J. G. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal of Control*, 44(4):943–976, 1986.

[60] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *World Congress on Formal Methods*, 1999.

[61] A. Ulusoy, M. Marrazzo, and C. Belta. Receding horizon control in dynamic environments from temporal logic specifications. In *Proceedings of Robotics: Science and Systems*, Berlin, Germany, June 2013.

[62] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[63] E. M. Wolff and R. M. Murray. Optimal control of nonlinear systems with temporal logic specifications. In *Proc. of Int. Symposium on Robotics Research*, 2013.

[64] E. M. Wolff, U. Topcu, and R. M. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 5018–5025, Karlsruhe, Germany, May 2013.

[65] K. W. Wong, R. Ehlers, and H. Kress-Gazit. Correct high-level robot behavior in environments with unexpected events. In *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014.

[66] T. Wongpiromsarn. *Formal Methods for Design and Verification of Embedded Control Systems: Application to an Autonomous Vehicle*. PhD thesis, California Institute of Technology, 2010.

[67] T. Wongpiromsarn, A. Ulusoy, C. Belta, E. Frazzoli, and D. Rus. Incremental temporal logic synthesis of control policies for robots interacting with dynamic agents. Technical report, March 2012.